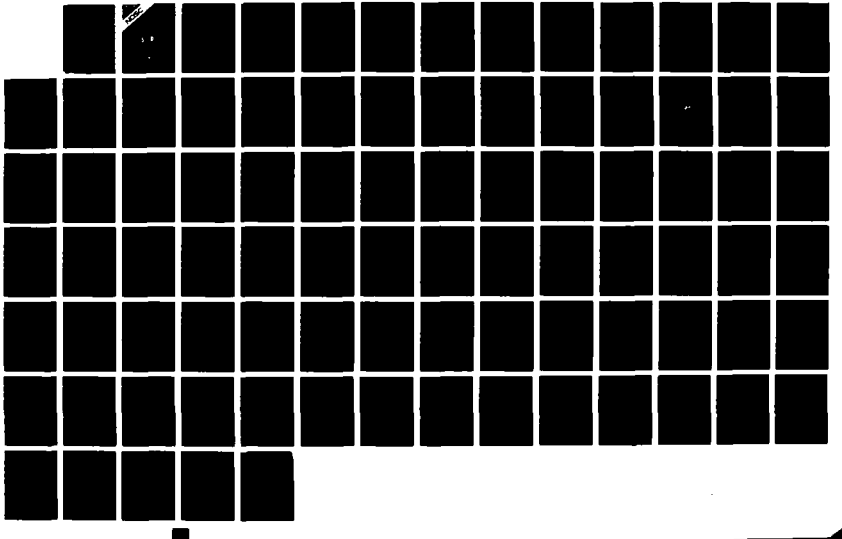
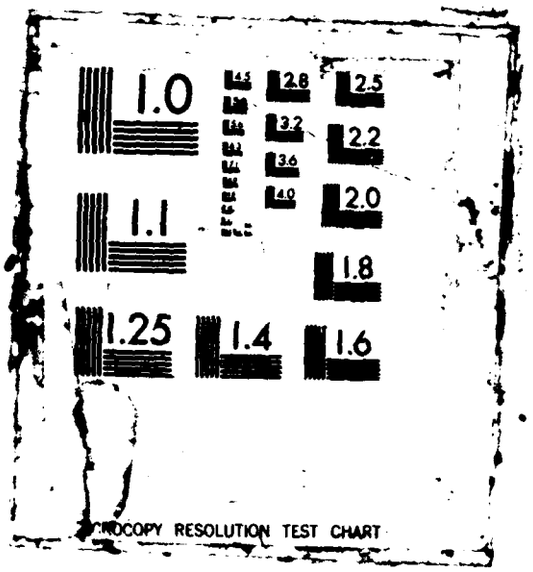
AD-A180 285 MACHINE SELF-TEACHING METHODS FOR PARAMETER OPTIMIZATION(U) NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA R A DILLARD DEC 86 NOSC/TR-1039 1/1

UNCLASSIFIED F/G 12/9 NL




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
ROCOPY RESOLUTION TEST CHART

(12)

NOSC TD 1039

NOSC TD 1039

NOSC

NAVAL OCEAN SYSTEMS CENTER San Diego, California 92152-5000

AD-A180 285

Technical Document 1039
December 1986

Machine Self-Teaching Methods for Parameter Optimization

Robin A. Dillard

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A180 325

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NOSC TD 1039	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Ocean Systems Center 6b. OFFICE SYMBOL (If applicable) Code 444	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) San Diego, CA 92152-5000	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO: Locally Funded; PROJECT NO; TASK NO; AGENCY ACCESSION NO

11. TITLE (Include Security Classification)
Machine Self-Teaching Methods for Parameter Optimization

12. PERSONAL AUTHOR(S)
Robin A. Dillard

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Oct 1984 TO Aug 1986	December 1986	89

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Learning systems, Artificial intelligence, System optimization, Radar, Control doctrine,

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The problem of determining near-optimum parameter-control logic is addressed for cases where a sensor or communication system is highly flexible and the logic cannot be determined analytically. A system that supports human-like learning of optimum parameters is outlined. The major subsystems are (1) a simulation system (described for a radar example), (2) a performance monitoring system, (3) the learning system, and (4) the initial knowledge used by all subsystems. The "initial knowledge" is expressed modularly as specifications (e.g., radar constraints, performance measures, and target characteristics), relationships (among parameters, intermediate measures, and component performance measures), and formulas. The intent of the learning system is to relieve the human from the very tedious trial-and-error process of examining performance, selecting and applying curve-fitting methods, and selecting the next trial set of parameters. A learning system to design a simple radar meeting specific performance constraints is described in detail, for experimental purposes, in generic object-based code.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Robin A. Dillard	22b. TELEPHONE (Include Area Code) (619) 225-7778
	22c. OFFICE SYMBOL Code 444

DD FORM 1473, 84 JAN

83 APR EDITION MAY BE USED UNTIL EXHAUSTED
ALL OTHER EDITIONS ARE OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

CONTENTS

INTRODUCTION

Increasingly complex sensor systems and communication systems are being developed; as the flexibility of systems grows, so does the need for automated methods of controlling them. Artificial Intelligence (AI) techniques exist now for building some parts of a sensor control system. We envision that the control logic can be implemented in the form of rules, but whatever knowledge representation is found to be best, the greatest difficulty will probably be in acquiring the optimum set of rules or control logic for a highly flexible system. For this reason, a learning system will be needed to aid in determining optimum mode choices, threshold settings, etc. The learning system would operate on feedback from a performance monitoring of simulated results and, later, from a performance monitoring of the sensor or communication system itself. Techniques exist for simulation and rule evaluation, but we found no learning techniques easily adapted to parameter control applications. The ability to learn new control rules or logic by self-teaching is the key feature of the envisioned automatic parameter control system, and our objective is to devise techniques leading to this learning capability.

The same parameter optimization techniques needed for a learning system of this kind can be used also to find optimum parameter values when designing a sensor or communication system. When selecting a simple example to treat in detail, one for which optimization can also be determined by analytical methods for verification, we found a design problem more suitable than a control problem. A learning system to design a simple radar meeting specific performance constraints is described in this report in generic object-based code.

One method of automating the learning process (and the approach primarily followed in this report) is to implement procedures that are humanlike, but which relieve the human from the very tedious trial-and-error process of repeatedly examining performance and intelligently selecting the next trial set of parameters or of rules controlling parameters. The intent is not to replace the methods of optimal control theory, but to augment them with AI reasoning techniques; e.g., to select and apply appropriate curve-fitting methods.

Another approach taken in this project is to determine the usefulness of several learning techniques that do not primarily model human thought processes. The results of the latter investigations will be reported in a separate document. In that study, we address much broader issues concerning the applicability of AI to system optimization problems.

APPLICATIONS

The kind of sensor system probably most in need of automatic parameter control is a radar system employing a phased-array antenna. In addition to the ability to point beams in rapid succession in a number of different directions, it may also have the ability to rapidly vary power, pulse duration, pulse repetition frequency (PRF), etc.

An example of a control problem for a very flexible (and hypothetical) radar is the following.

Environment	-->	Optimum modes/parameters	
Track data (dynamics, cross sections, IDs, etc.)		Next pointing angle(s)/angular coverage (1 pencil? 1 fan?, n simultaneous? combination?)	
Weather, terrain		Per beam:	PRF
Other sensor tracks			dwell time
Intelligence			waveform/resolution
ECM			frequency
Reaction times			power
Etc.		Etc.	

Knowledge about a target, such as its identity (ID) being hostile, can influence the parameter selection. Weather and terrain here include not only propagation anomalies and land topography, but sea state, drift ice, and other natural phenomena. Intelligence can include sighting reports from other ships, aircraft, or satellites, and reports of expectations of certain activities. Electronic countermeasures (ECM) include jamming, chaff, and deception.

There are many diverse kinds of intercept receivers. Some have in common with conventional radars the feature of rotating to cover in azimuth, and most scan in frequency over a certain band of frequencies or over several bands. More recent designs permit an agility in frequency scanning analogous to that of beam pointing. While some intercept receivers could profit from automatic parameter control, the greatest need for control will probably be for intercept systems that incorporate several receivers; e.g., a warning receiver, an analysis receiver, and a direction-finding receiver. In this application, the learning methods should be applicable both to the problem of individual parameter control and the problem of the division of utilization. As with radar problems, judgment of intercept performance will largely be based on detection probability, false-alarm rate, and resolution.

Many of the same propagation anomalies affecting radar also affect the intercept receiver. In place of the target environment, with the many physical constraints on targets, we instead have the problem of a complex variety of signals. Since the number of possible signal scenarios is unlimited and the likely scenarios will constantly change, the learning process would have to continue always, rather than converge to one that will be adequate for a long period.

Communication systems also have propagation conditions, jamming, other-user noise, etc., to contend with, but the detection problem is very different since they are detecting known, friendly signals. Other kinds of environmental features are the state of EMCON (EMission CONtrol), various security requirements, message priority, and traffic requirements. Generally, there will be fewer parameters to control, and these typically would be frequency, power, and the modulation and coding scheme. In many cases, analytical optimization will be possible, and a learning system would be unnecessary.

There are two categories of problems, as mentioned earlier. One is the optimization of a fully described system for each of the various scenarios it is likely to face. The other is the optimal design of a system, given these scenarios. In the latter case, optimization may be needed over all scenarios, so the process can be much more time-consuming. Basically, the same kinds of learning techniques are needed for both kinds of problem.

PARTICIPATING AI SUBSYSTEMS

Figure 1 is a block diagram of one concept of an AI system for controlling the parameters of a radar. In an operational system, a radar and the actual environment would replace the simulator. The control logic learned in the simulation stage is refined in the operational environment. The function of the parameter-control system is to reset the parameter and mode settings appropriately as the situation changes. While a control system should be highly automated, some amount of operator control will be useful and necessary.

Knowledge common to a number of sensor systems would be built into the knowledge base initially, and a data/knowledge acquisition system would obtain from a human a description of the particular system to be optimized and his performance specifications, and would restructure these into the system syntax. The knowledge-base box in figure 1 includes initial knowledge and learned knowledge.

The radar, tracker, targets, and weather effects would be simulated. The simulation system would probably resemble ROSS, a high-level AI programming language developed by the Rand Corporation specifically for warfare simulation [1]. In the arrangement shown, the radar parameter values used in the radar simulation are provided by the rule-based system. (While a rule-based system is envisioned, we may find some other kind of programming to be more appropriate.) The function of the rule-structuring and organizing system shown in figure 1 is to organize the mappings of environment to parameter selection in a useful form. It might use techniques such as those of BACON* [2, 3] to fit curves to the data. (The other study under this project looks at ways of learning control rules directly, in which case the rule structuring and organizing system would be an integral part of the learning system.)

The learning system described in this report provides trial parameter values directly to the simulator, rather than via a structuring and organizing system and a rule-based system. A structuring and organizing system would be needed later, but the design of such a system is not an objective of this project.

---

* Named for Sir Francis Bacon, in honor of his theory of induction.

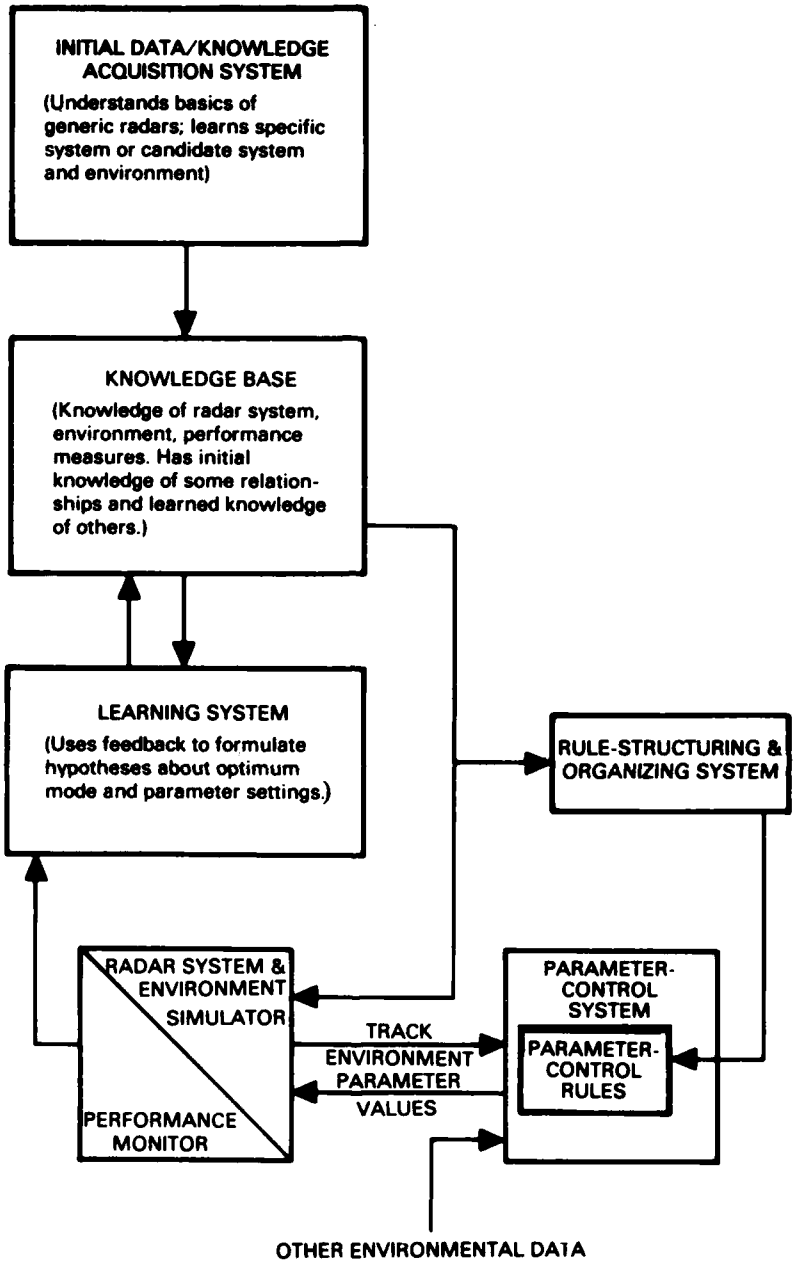


Figure 1. Overview of a system for building a set of parameter-control rules for radar.

Similarly, development of the simulation system is not a part of this effort, and a crude substitute would serve any needs during experiments with a learning system. Figure 2 illustrates how early experiments can be conducted with a simulator, however crude it is. Verification of the techniques for a very simple radar system is possible by computing performance measures by means of well-known radar formulas.

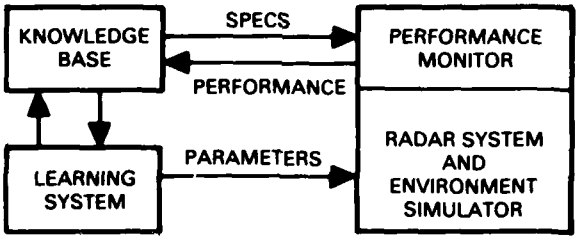


Figure 2. Learning system interaction in an early stage.

PERFORMANCE MEASURES

The simplest approach to assigning an overall measure of performance resulting from a set of parameter values is to measure each of the various kinds of performance and to use the sum or weighted sum. Examples of sources of component measures for a two-dimensional agile-beam radar are false-track rate, average expended energy, and track quality. There should be several track-quality measures: e.g., one for each combination of range and speed; for long, medium, and short range; and for fast, medium, and slow targets. The track-quality measures could further be broken into component measures relating to detection probability and resolution.

We have found that a performance measure expressed in "units of rejection" is more convenient than one in "units of goodness." Figure 3 shows how a measurement of the false-track rate would result in a component measure, "false-track_units," expressed in units of rejection.* For many of the measures, the units will be decreasing rather than increasing as shown. If the number of units exceeds the maximum allowed for that measure, the candidate parameter set is rejected. The amount in excess can help determine the next set of parameters to try.

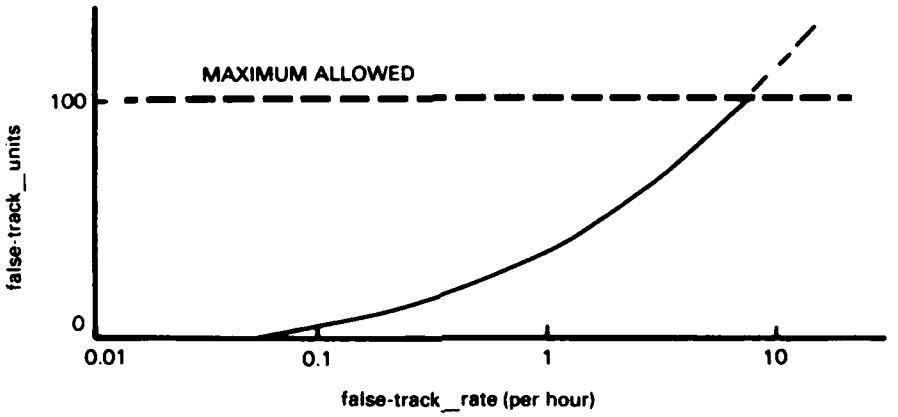


Figure 3. Example of a component measure of performance.

While it is convenient to compute an overall performance measure by summing the component measures, and then to select the parameter set yielding the minimum value of the overall measure, it is very likely that some of the component measures will be low at the expense of some being high. As discussed in the next section, we will try to force the component measures all to have approximately the same "rank" on their range of units of rejection.

---

*Underlines connect words when the combination is to be a single word in a computer program. Hyphens serve their normal function, but are sometimes disallowed in computer words.

THE OPTIMIZATION PROBLEM

MINIMIZING AND BALANCING THE PERFORMANCE MEASURES

Denote the jth variable parameter or mode as P_j, and denote a particular value of P_j as p_j. Each set $\{p_j\}$ of parameter values results in a set $\{M_i\}$ of component measures, and the overall performance measure is overall_measure = $f(\{M_i\})$.

When simulation is not prohibitively expensive, the optimization process should begin with a "coarse scan." The first coarse scan would be for the average of the environmental situations to be considered. The "surface" created [in (n+1)-dimensional space for n parameters] by mapping overall_measure as a function of $P_1,\ldots,P_n$ should have one or more "valleys." (The remaining discussions are easiest to visualize for n = 2.) Since the surface will be disjoint wherever a P_j changes its value if P_j is discrete, and since some parameters have non-numerical values, this characterization is not accurate but should convey the idea.

If several valleys occur, all roughly the same depth, all should be investigated for use, since the random use of parameters makes it more difficult for the enemy to predict the system's behavior or to interpret it. If the first coarse scan is for an average situation and yields only one valley, it is probably wasteful to make a coarse scan for other environments. Instead, that valley plus knowledge of how the environment affects performance can be used to estimate where the valley is for the other environments.

Defining the boundary points of a valley found in the coarse scan is useful mainly as a step in determining whether there are other valleys. If a low value of overall_measure is found outside of the deepest valley (deepest known after only a coarse scan), another valley is defined there. The valley, or at least one of the valleys, should contain the minimum overall_measure. (A valley's size would depend partly on the coarseness of the scan.) Simple algorithms can be used to approximately define this region; for example, the "three-tier" method described in appendix A.

When simulation is too costly to permit performance of a coarse scan, a "zero-in" method can be used. The risk in using this method is that, if there is more than one valley, the deepest may not be the one found. Descriptions of both the coarse-scan method and the zero-in method are given in appendix A.

Once a coarse valley is found (either by a coarse scan or by a zero-in search confined to the same increment sizes), curve fitting and other numerical methods are used to search for the true minimum. After the valley minimum is found, the next stage of optimization takes into account the values of the component performance measures and works to balance them (i.e., to give them roughly the same rank on their respective scales), while not increasing overall measure by a significant amount. Knowledge of how each performance measure is affected by the variable parameters guides the selection of the next trial set of parameters. Details of a way to do this are given in appendix A for the example problem. If there are multiple valleys, a valley in which (or near which) the performance measures can be balanced may be preferable to a somewhat deeper one where they cannot be. Experiments are needed to determine how workable the balancing concept is.

As we will see in the next section, this characterization of the optimization process is oversimplified for many applications. For example, certain parameters of a flexible agile-beam radar system can vary from beam to beam. It is highly impractical to decide, after each dwell of the antenna beam, which parameter set to use next. Instead of a single value of dwell time, for example, we might assign the next-scan dwell times: 90 ms in beams [3,20,44], 60 ms in beams [5,9,31,57], 30 ms in all other beams. Next, we consider some alternative methods of planning ahead.

HOW FAR AHEAD TO SCHEDULE

Consider the control problem for a two-dimensional, agile-beam radar. Should the parameter-control system decide each parameter change just prior to the time of the change? Or, should it plan ahead the pattern of changes over some period of time? Some of the possibilities for decision times are

- Per beam position,

- Per m beam positions,
- Per sector (fraction of total azimuth range),
- Per scan (or over total azimuth covered),
- Per time unit.

Similar decisions need to be made for other radar types and for other sensors. In addition, it may be less satisfactory to specify every parameter during the next period than to allow the plan to change automatically as a result of data obtained. For example, the sudden occurrence of jamming may call for a change in the plan. The radar data obtained in a beam can affect the plan for that scan; e.g., doubling the dwell time or the power if a target-present decision occurs in a beam where none is expected.

NEXT-SCAN PLANNING

Here we consider ways of planning ahead one scan for agile-beam radars. In general, more attention should be given to those beam positions where detections are likely to occur for a current track or where intelligence or other sensors have indicated a likelihood of an approaching target. Hostile, high-speed, and maneuvering targets should receive additional attention. For fast or maneuvering targets, this attention is likely to consist of more frequent looks. A high-resolution waveform may be used when multiple targets in a beam need to be resolved. For long-range or small targets, an increase in power or an extra-long dwell time may be appropriate. A high-priority search may call for high power, extra-long dwell time, or more frequent looks.

A candidate doctrine can be formulated and experimentally refined for choosing the next scan's parameters. This doctrine will probably be in the form of a set of rules. The doctrine will necessarily be different for high-target-density cases than for low-density, and will require additional flexibility to adjust to the exact density. For example, the sum of the dwell times needed in each beam may exceed the maximum

scan time desired, and modifications to the first cut will be needed, such as reducing slightly the time for each or using high power rather than extra-long dwell times.

A simple approach that can easily be implemented in the form of rules is to specify what the values of the various parameters will be for each beam in the next scan, based on the situation in that beam. This technique is usable if the geometry is such that the target can change, at most, one beam position per scan. Tracker extrapolation is needed for best results. Table 1 gives an example of how the problem of specifying values of parameters $\{P_j\}$ for each beam and scan is converted into the problem of selecting values of parameters P_{jk} that relate to situation k. The following set of situations are assumed in the example in table 1. One or more of these situations holds for each beam on each scan.

0: DEFAULT. No track is likely to continue in that beam on that scan, and there is no indication from intelligence or ESM that a target is likely to enter from that direction.

1: TRACK CONTINUATION. A current track could continue in or enter that beam. Sometimes there will be two or more beams having a likelihood of containing the next report of that track, especially for a high-speed target.

2: HIGH SPEED. The target that could be in that beam is traveling at a high speed.

3: WEAK. The signal strength of the target's echoes is very weak, either because of the target's distance or its size.

4: CLOSE RANGE. The target is likely to be too close to use a high-resolution pulse. (The duration of the pulse results in a minimum range for observing targets.) Another less-close category may be desirable for targets that are so close they need extra surveillance but are beyond the minimum range.

5: PRIORITY SEARCH. Intelligence or other sensors have indicated that a target might be entering from that direction at any time.

6: TWO IN ONE BIN. There is a likelihood that at least two targets are in the same range bin, in that beam.

Table 1. Example of per-beam parameters for an agile-beam radar. (One "scan" is one complete coverage.)

Situation in Beam on That Scan	Number of looks P1 [integer]	Dwell Time P2 [ms]	Power P3 [kW]	Pulsewidth P4 [μs]
0 Default	P10 = 1	P20	P30	(low resolution) P40
1 Track Continuation	P11	P21	P31	P41
2 High Speed	P12	P22	P32	P42
3 Weak	P13	P23	P33	P43
4 Close Range	P14	P24	P34	P44
5 Priority Search	P15	P25	P35	P45
6 Two in One Bin	P16	P26	P36	(high) P46
	$P_j = \max_k P_{jk}$			Low and high could be alternated in some situations.

TWO-DIMENSIONAL AIR RADAR EXAMPLE

The pertinent specifications of this hypothetical radar system are

- Agile fan beam, 10-degree beamwidth,
- 162-,nmi instrumented range,
- 300 pulses per s,
- Pulse duration = 100 μs (low resolution)
  or 2 μs (high resolution),

- Dwell time per beam = 0.1 s (30 pulses)
  or 0.2 s (60 pulses),
- Looks (beams) per beam position per scan = 1 or 2.

A "scan" is completed when each beam position has had at least one look. The beam positions are selected pseudorandomly, but a second look can be selected algorithmically to space it about half a scan in time from the first look.

The per-beam-position variables (per scan) are

- Looks per scan = 1 or 2,
- Dwell time (per look) = 0.1 s or 0.2 s,
- Pulse duration = 2 μs or 100 μs.

Using the earlier definitions of beam situations, we have the following initial knowledge of desirable constraints on the control rules. All parameters left unspecified are to be learned.

Situation 0 (default): 1 look, 0.1-s dwell, 100-μs pulse.
Situation 1 (track continuation), unless situation 4 or 6 is true: 100-μs pulse.
Situation 2 (high speed): 2 looks.
Situation 3 (weak): 0.2-s dwell and probably 2 looks.
Situation 4 (close range) or situation 6 (2 in 1 bin): 2-μs pulse.
Situation 5 (priority search): 2 looks and/or 0.2-s dwell time.

Either the rule conditions would need to include target density considerations, or several rule sets would be needed, each for a different range of target count.

THE LEARNING TECHNIQUES

The learning methods considered in this report are humanlike; they are intelligent trial-and-error procedures employing numerical methods and common-sense reasoning. To simplify the discussion, assume that a set of parameter values $\{p_j\}$ (or $\{\{p_{jk}\}\}$) lead to the performance measures $\{M_i\}$ and an overall measure M. For the system

outlined in appendix A. M is the sum of the values of M_i. The reasoning procedures outlined in appendix A rely on "dependencies." The kind of dependency used there is a relationship between a variable parameter p_j and a performance measure M_i. Typically, the relationship type is "increasing" or "decreasing." Two other kinds of dependencies also can be used: the relationship can be between a variable parameter and an "intermediate measure" (e.g., detection probability) or between an intermediate measure and a performance measure. (The latter two kinds of dependencies can be used by the system to generate dependencies between variable parameters and performance measures, rather than have the user provide them.) Examples of dependencies are given in the next section. The following procedures are used in the system outlined in appendix A for a simple mechanical-scan radar.

- During the valley-finding process, a "boundary_checker" determines whether a performance measure M_i exceeds its maximum allowed value. If it does, an "inbound_direction" structure is created that lists the parameter-change options. This is done both for the coarse-scan method and the zero-in method, but is needed more in the latter case.

- During the "balancing" of the component performance measures (the attempt to avoid having one performance measure low at the expense of another being high), the "balancer" creates a "reduction_direction" for the performance measure highest on its own scale. This structure is similar to the "inbound_direction" in that it lists parameter-change options.

- The zero-in method of finding a coarse valley uses humanlike reasoning to compare values of M among past samples and to use the results to decide the "direction" in which to move for the next sample.

- Humanlike procedures are also used to select and apply curve-fitting operations in the search for the minimum value of the overall_measure M within a coarse valley.

A kind of reasoning that should be experimented with at a later stage would use what we call an "unknown_dependency." For example, the relationship between

radar scan rate and the average number of detections per minute (the "hit rate") is not known because the number of detection opportunities increases with scan rate, but the detection probability per scan decreases as a result of the reduced dwell time per beam. The system could, after a few samples, hypothesize the relationship and use this relationship to converge faster to the optimum set of parameters.

Humanlike reasoning, to interpolate or extrapolate among scenario results to find initial parameters for a new scenario, should also be implemented. For example, the best combination of parameter values against a medium-speed target is likely to lie (respectively) between the best for a fast target and the best for a slow target. The best combination for a moderate number of targets should be somewhere between those for high-density and low-density target situations. Similarly, the best combination for a signal intercept system when a moderate number of signals is present should be between those for high signal density and those for low signal density.

AN OBJECT-ORIENTED MODEL

OBJECT-ORIENTED PROGRAMMING

An object is a package of information and descriptions of its manipulation [4,5]. Objects are in a hierarchy, and the primary use of the hierarchy is inheritance of attributes--each object inherits the attributes and procedures of its parent object. The action in object-oriented programming results from "message passing" among the objects.

INTERACTION OF SUBSYSTEMS

A large overlap occurs in the initial knowledge and dynamic knowledge required by the sensor simulation system (e.g., radar and tracker simulation) and the learning system. This overlap is a strong argument for implementing both in the same high-level language. Simulation is best accomplished with object-oriented programming, as opposed to rule-oriented programming or procedure-oriented programming, but, fortunately, the object-oriented approach appears to be best also for a learning system

of the kind proposed. Some high-level languages allow combinations of object-oriented and rule-oriented programming, which means that testing of the parameter-control rules (the product of the learning system and a rule-structuring and organizing system) in the same expert system is feasible. While this project is concerned mainly with the design of learning techniques, and does not address rule-structuring and organizing, the learning techniques must be compatible with these other processes.

Figure 4(a) shows the basic structure of a system for learning radar parameter control rules, and 4(b) shows in detail an example of the initial knowledge needed by the subsystems. The initial knowledge shown would vary with the problem. For example, when the objective is to design a radar system, there would be two types of variable parameters--the design variable, which would stay fixed over all scenarios, and the system variable, which generally could be varied from beam to beam or from scenario to scenario.

Not shown in figure 4(b), but important to the learning process, are a number of intermediate variables or measures. Examples of intermediate measures for an agile-beam radar are average scan duration and average energy per scan.

The representation of the intermediate measures and performance measures are under the hierarchy of the object initial_knowledge. The following are examples of dependencies:

```
approx_dependency_1
    type = against
    quantity1 = threshold
    quantity2 = detection_probability

approx_dependency_2
    type = against
    quantity1 = threshold
    quantity2 = false-alarm_probability
```

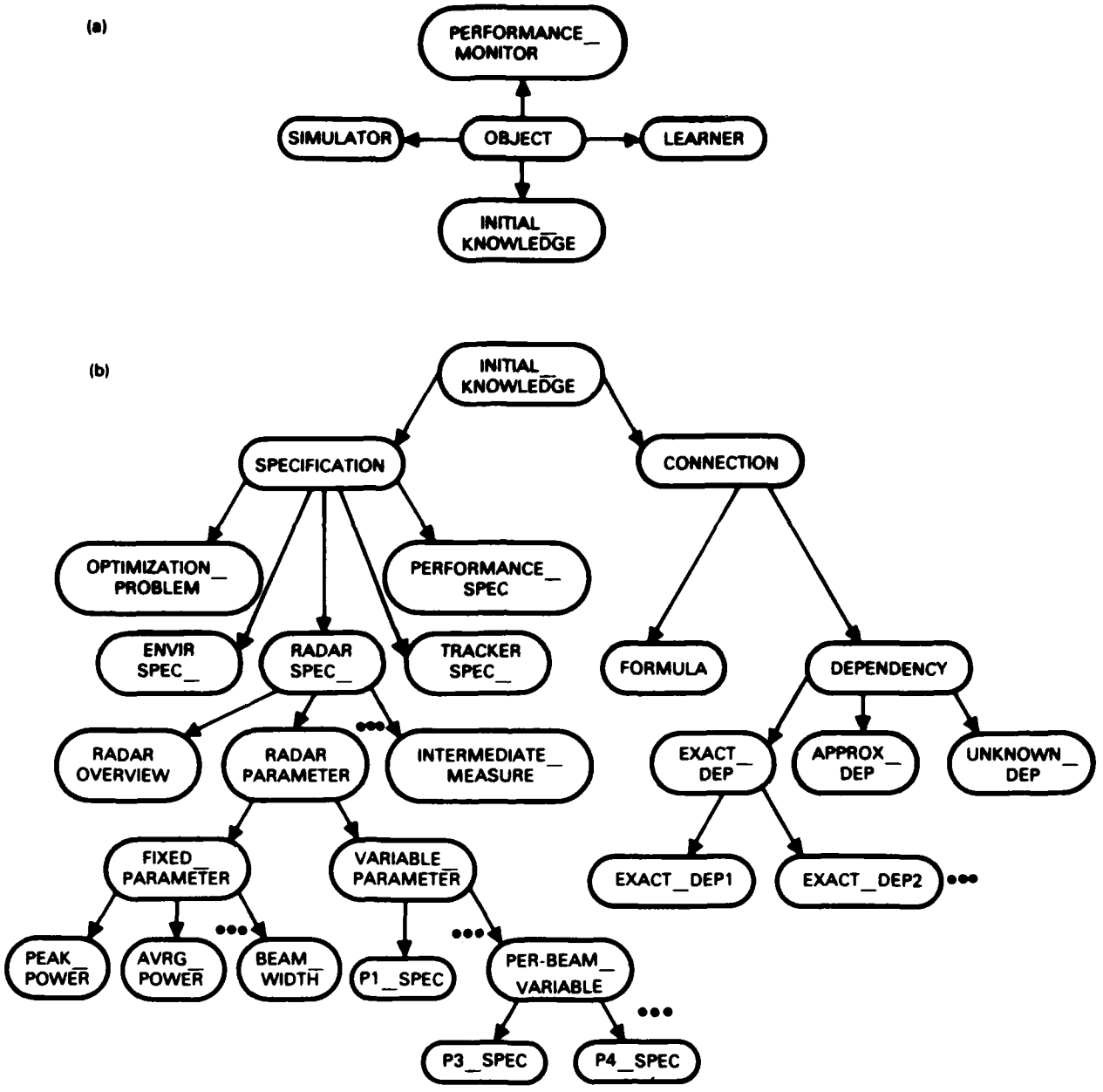


Figure 4. Object-based system for radar parameter optimization. (a) High-level overview. (b) Example of hierarchy in initial knowledge for an agile-beam radar.

exact_dependency_1
    condition = CFAR [Constant False-Alarm Rate]
    type = proportion
    quantity1 = false-alarm_probability
    quantity2 = false-alarm_rate

Examples for an agile-beam radar:

approx_dependency_3
    condition = agile_beam
    type = support
    quantity1 = default_dwell_time
    quantity2 = average_scan_duration

approx_dependency_4
    condition = agile_beam
    type = weak_support
    quantity1 = sit_3_dwell_time
    quantity2 = average_scan_duration .

Examples for designing a mechanical-scan radar:

exact_dependency_2
    condition = mechanical_scan
    type = inverse_proportion
    quantity1 = scan_rate
    quantity2 = pulses_per_beam

unknown_dependency_1
    condition = mechanical_scan
    type = mixed
    quantity1 = scan_rate
    quantity2 = hit_rate .

(Hit rate is the product of detection probability and scan rate, but detection probability decreases as scan rate increases.)

The following is an example of a formula:

```
formula_1
    type = increment
    quantity = energy_units
    calls = (power pulse_duration)
    incr_interval = per_pulse
    incr_duration = scenario
    incr_size = (power * pulse_duration) .
```

To use this formula intelligently, some object, perhaps called "mathematician," should have a procedure for using the knowledge that multiplying by n is equivalent to summing over n pulses when the power and pulse duration remain constant.

Formulas would also be used by the simulator's umpire to decide whether a signal threshold has been exceeded. The actual representation of a formula would vary widely among different object-oriented languages, and is unlikely to be in the simple form shown. The implementation would involve messages; e.g., at each beam pointing, a message would be sent: (tell energy_units increment your value by (tell mathematician compute (ask formula_1 recall your incr_size))). The representation of the computation is more likely to be a LISP function (if procedures are written in LISP) than a slot value as shown.

THE SIMULATION SYSTEM

A simple scheme for simulating targets for a two-dimensional radar and for constant course and constant speed is shown in figure 5. The beam position parameters in figure 5 are

$$O = 2\pi/n$$

$$xk = r \cos k\theta$$

$$yk = r \sin k\theta$$

and the beam/target intersection parameters are

$$x' = (cxi - yi)/(c - yk/xk)$$

$$y' = x'(yk/xk)$$

$$r' = \sqrt{(x')^2 + (y')^2}$$

$$t' = t + v\sqrt{(x' - xi)^2 + (y' - yi)^2} \quad .$$

A target is created by randomly generating an entrance time (uniform between simulation start time and simulation end time) and two beam positions (each an integer uniform between 0 and b - 1, for b beams), one for entrance and one for exit. The entrance and exit points are labeled (xi,yi) and (xj,yj), respectively.

Each track will consist of a number of positions of the form (beam number, range-bin number, time-in, time-out), and will have associated with it a speed and a cpa (closest point of approach). (The track actually will be valid for various combinations of speed and range-bin size.) Some tracks can be delayed versions of others; i.e., additional tracks can be generated simply by adding a time constant to the time-in and time-out, if computation time is expensive. A number of such tracks for a variety of speeds would initially be generated, and each could be used with any cross section consistent with its speed. For different scenarios, different combinations of varying numbers of tracks would be chosen.

If the radar system allows the occasional use of a high-resolution waveform, a temporary modification to the track sometimes will be needed. Rather than store such extensive data in track form, it would be better to compute them upon demand. For example, whenever two targets appear to be in the same range bin, the exact range could be computed for each. If the ranges differ by more than the range resolution of the high-resolution pulse, the targets would be declared resolved.

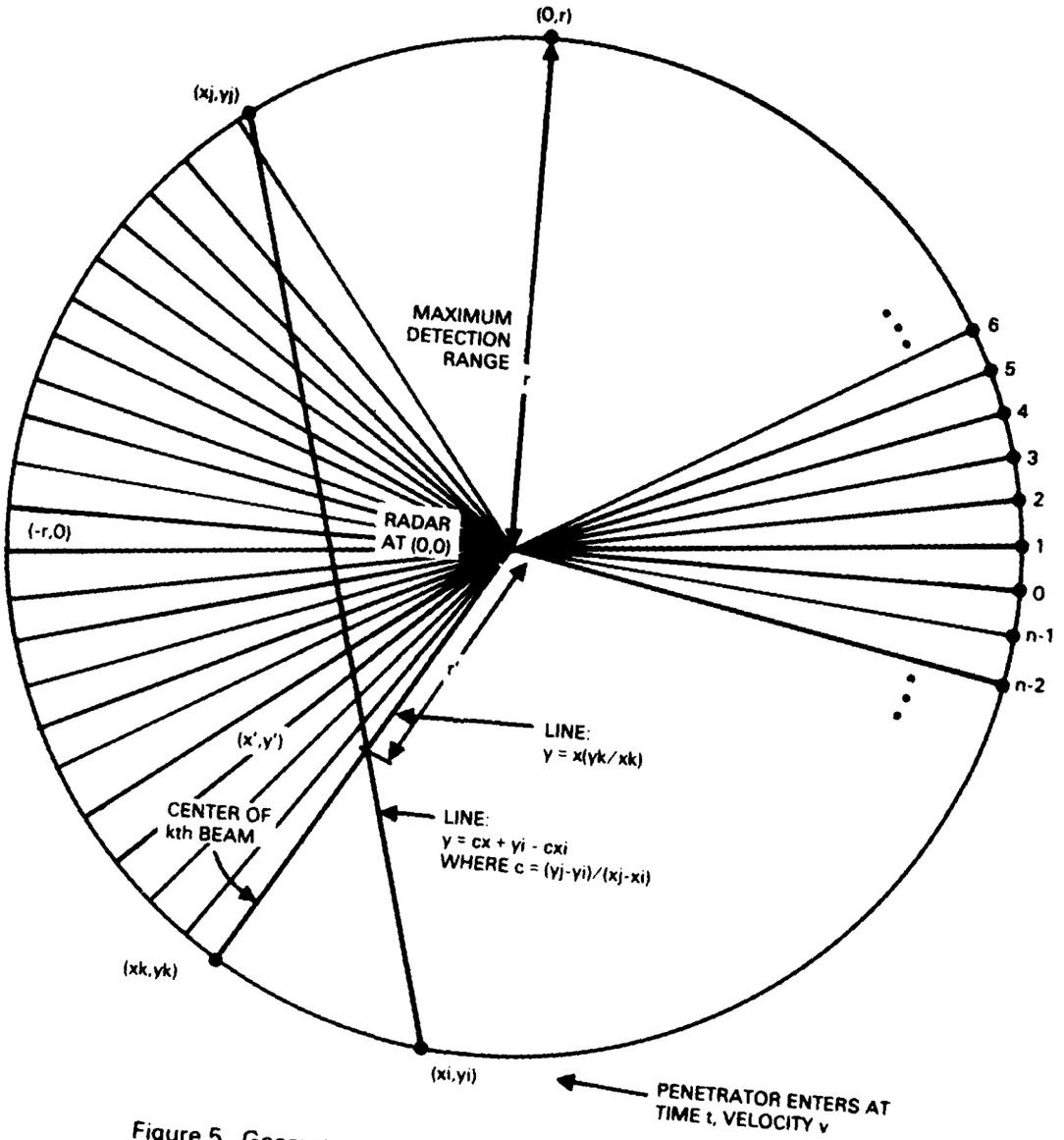


Figure 5. Geometry for a two-dimensional radar simulation.

One scenario [e.g., one set of tracks and one set of assumptions about weather (the simulator's "umpire" uses weather assumptions in generating signal-present decisions)] would be run over and over, as different radar parameter sets or control rules are tried. Since clutter and receiver noise introduce randomness to the detection process, identical experiments sometimes will have to be repeated many times, with detection differences resulting from a random-number generator used in the umpire module. Additional runs using random variations in the same basic scenario would be useful in the final stages of optimization; e.g., using variations having the same number of tracks and the same distribution of speed and of cpa.

If the system is of the CFAR (constant false-alarm rate) type, which frequently is the case, there is no need to have the simulator's threshold umpire decide whether signal is present for every range bin of every beam. Instead, empty bins can be randomly chosen as having signal decisions. This is also possible with a non-CFAR radar, although other factors such as clutter regions have to be considered.

A RADAR DESIGN EXAMPLE

THE PROBLEM

As a simple example to work with, we have chosen a radar design problem. The radar to be designed is a mechanical-scan, fan-beam, surface-search radar. It is a simple, low-power radar, with, perhaps, navigation or collision avoidance its primary purpose. Its beamwidth, instrumented range, and average power are specified, and the PRF (pulse repetition frequency), peak power, pulse duration, scan rate, and PFA (probability of false alarm) are to be chosen. There will be two sets of values of these "variable" parameters. One is a high-target-density mode, for use when traveling in a merchant lane or near a port. The other is a low-density mode for traveling in open seas outside of all merchant lanes. The detection probability formula models no real situation, but is about the simplest one having the correct properties; e.g., the computed detection probability will correctly increase or decrease with the parameters that actually affect detection probability. Details are listed below.

Fixed parameters

beamwidth
    units = degrees
    value = 1.5
instrumented_range
    units = nautical miles
    value = 25
average_power
    formula = peak_power * pulse_duration * PRF
    units = watts
    value = 10.

Variable parameters

PRF
    units = pulses per second
    minimum value = 500
    maximum value = 2100
peak_power
    units = watts
    maximum value = 1.0E4
pulse_duration
    units = seconds
    minimum value = 0.5E-6
    maximum value = 1.0E-5
scan_rate
    units = revolutions per minute
    value = 6
    maximum value = 18
PFA
    units = probability (per range-bin decision).

We are assuming that energy is exactly proportional to peak_power, PRF, and pulse_duration, and detection_probability is a function of energy_per_pulse (at the receiver) and pulses_per_beam. Since average_energy is a fixed parameter, the variable parameters peak_power and PRF are not varied independently in the optimization process. They are determined by the following rule: Use the smallest PRF such that (1) PRF >= 500, (2) pulses_per_beam = integer, and (3) peak_power =< 1E4 watts. An algorithm for this follows. (NLI = next lower integer.)

If pulse_duration > 2E-6,
    let i = 1 + NLI(125 / scan_rate);
otherwise,
    let i = 1 + NLI(1 / (4E3 * pulse_duration * scan_rate)).
Let PRF = 4 * i * scan_rate.

Intermediate measures

pulses_per_beam
    formula = (PRF * beamwidth) / (6 * scan_rate)
            = PRF/(4 * scan_rate)
    units = pulses per beam position per scan
range_resolution
    formula = 0.5 * pulse_duration * c [c = speed of light]
            = 0.8094E5 * pulse_duration
    units = nautical miles
cells_per_beam
    formula = instrumented_range / range_resolution
            = 25/range_resolution
units = resolution cells per beam
decision_rate
    formula = (6 * scan_rate/beamwidth) * cells_per_beam
            = 4 * scan_rate * cells_per_beam
units = signal or noise decisions per second

false_alarm_rate

formula = 3600 * decision_rate * PFA

units = false alarms per hour

energy_per_pulse

formula = peak_power * pulse_duration

= average_power/PRF

= 10/PRF

units = watt-seconds (transmitted)

detection_probability = exp((ln PFA) /

(2500/SQRT(scan_rate * PRF) + 1))

units = probability (of detecting "standard target") per scan.

A standard target here is a surface target having a cross section of 10 meters squared and a range of 10 nautical miles. The formula for detection_probability uses a Rayleigh approximation for noncoherent integration, where (see figure 6) signal-to-noise ratio = constant * SQRT(pulses_per_beam) * energy_per_pulse = 2500/SQRT(scan_rate * PRF).

hit_rate

formula = scan_rate * detection_probability

units = hits (detections) per minute per standard target.

Performance measures

Notes:
1. All have the units, rejection_units.
2. All have max_units_allowed = 100.
3. Figures 7–9 give plots of the performance measures.

FAR_units = 120 * exp(2 * ((log false-alarm_rate) - 2.1))

hit_rate_units = 115 * exp(-0.7 * (hit_rate - 3))

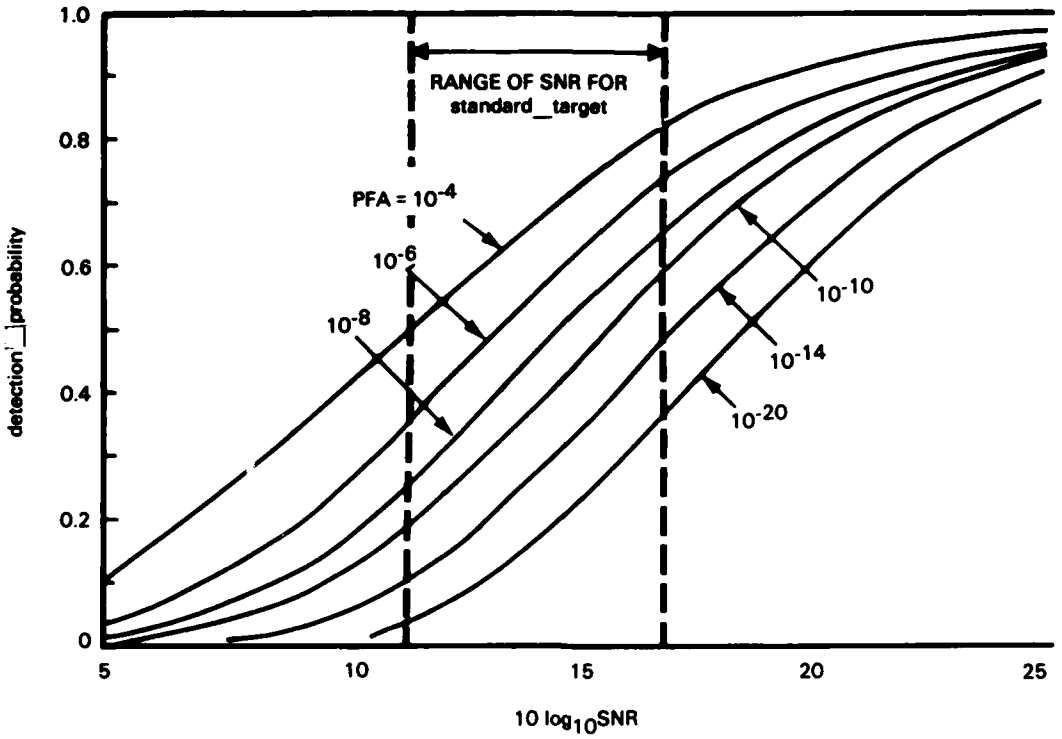


Figure 6. Per-scan detection probability versus 10 log SNR (where SNR is the signal-to-noise ratio) for a Rayleigh distribution of signal and noise. For average power = 10 watts and SNR = 2500/SQRT(scan_rate * PRF), SNR varies over the range shown as pulse_duration and scan_rate are varied.

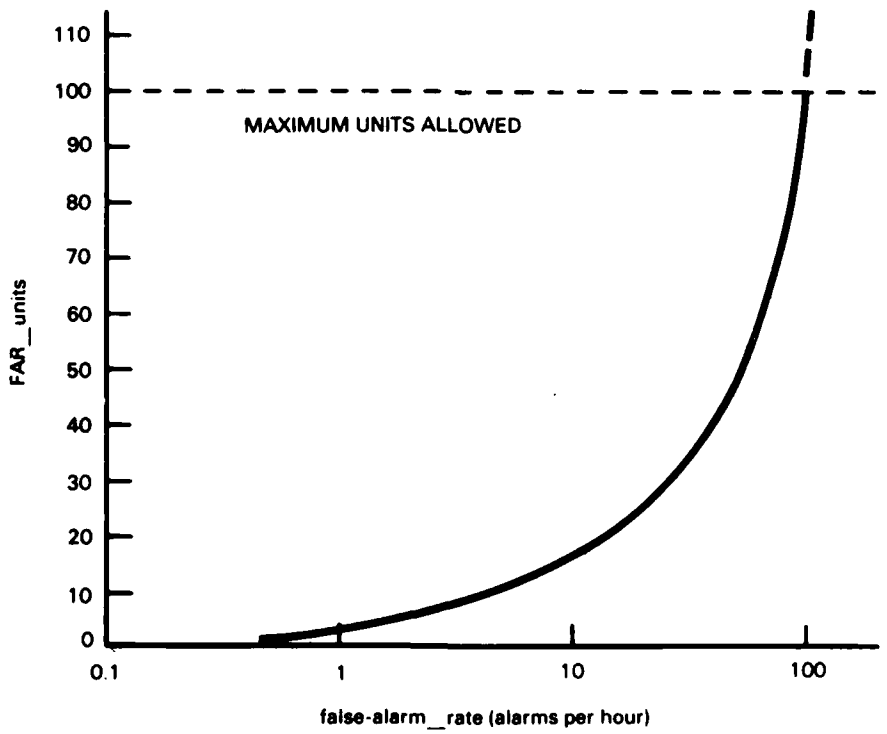


Figure 7. Performance measure FAR_units versus false-alarm_rate.

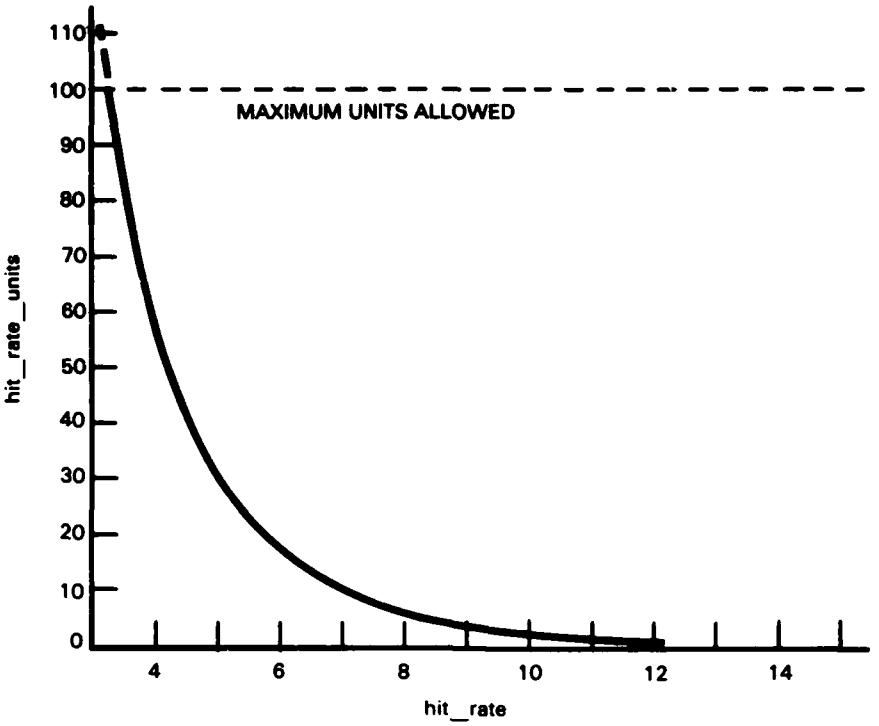


Figure 8. Performance measure hit_rate_units versus hit_rate, the expected number of times per minute a particular "standard target" is detected. Here, a standard target has a range of 10 nmi and a radar cross section of 10 m^2.

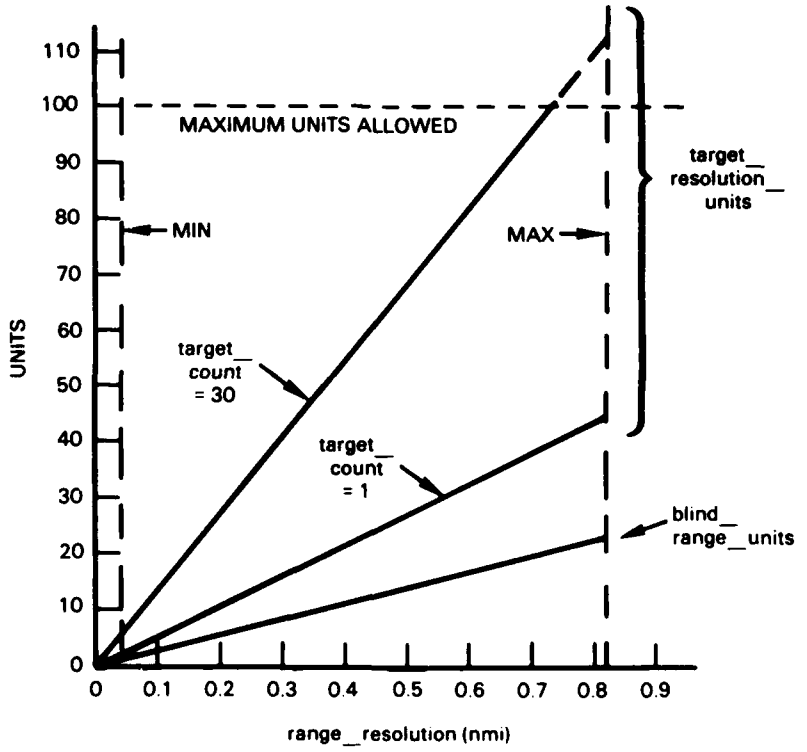


Figure 9. Two performance measures (target_resolution_units and blind_range_units), as a function of range_resolution.

target_resolution_units = (50 + 3 * target_count) * range_resolution.

(When two targets fall in the same resolution cell, they appear as one larger target. The problem worsens as the cell size increases and the number of targets increases. The target_count is the actual number of ships, except ownship, within the instrumented range.)

blind_range_units = 30 * range_resolution.

(The radar does not receive while transmitting; hence it is blind to targets at a distance less than the range resolution.)

overall_measure = FAR_units + hit_rate_units +
target_resolution_units + blind_range_units.

RELATIONSHIPS AMONG PARAMETERS AND MEASURES

Figure 10 shows which measures are affected by which parameters and measures. Note that a convenient alternative to computing an integer value of cells_per_beam from pulse_duration is to specify the integer value and compute range_resolution. From figure 10, it is not obvious that detection_probability, as defined earlier, can be computed directly from PRF and scan_rate, but doing this is a short-cut method of reducing computations. In a realistic formula for detection probability, the energy_per pulse and pulses_per_beam would be needed to estimate the loss attributable to noncoherent integration. Not shown in figure 10 is target_count, which affects the performance measure target_resolution_units.

The relationships listed below indicate whether a measure increases or decreases with a parameter or other measure, and when this change is linear.

Notation:

==> implies
=L=> implies linear relationship
↑ increases

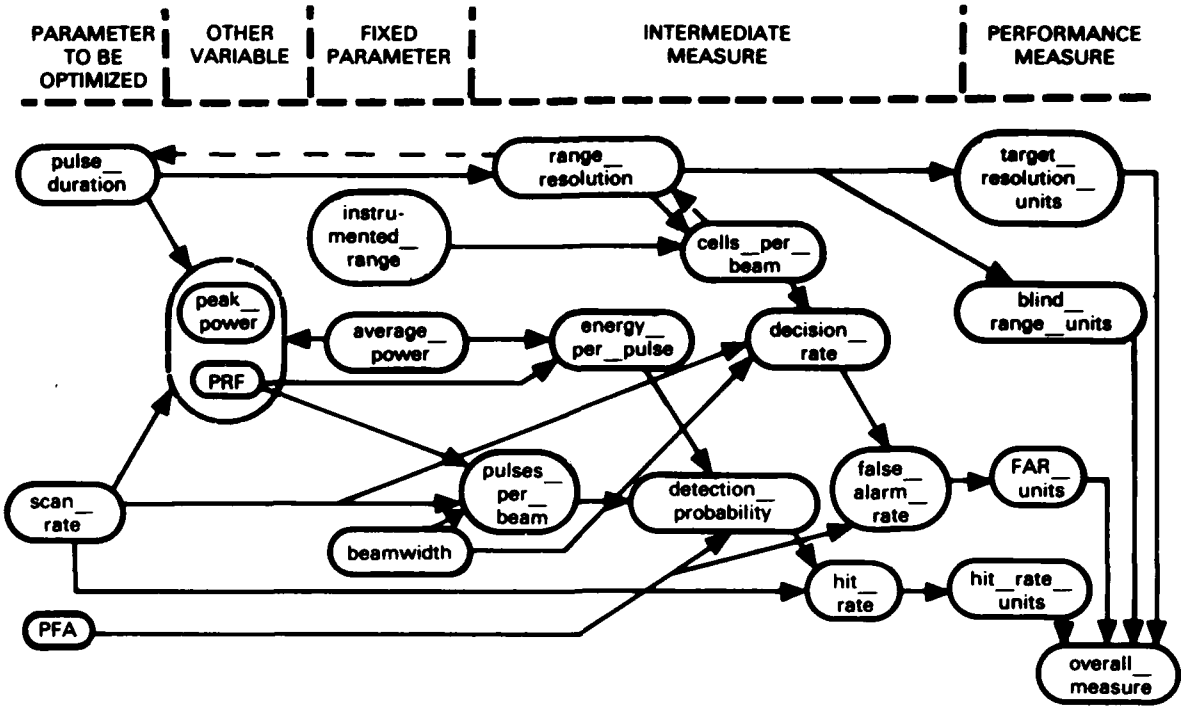


Figure 10. Flow diagram of parameter effect on performance measures, via intermediate measures. Instead of specifying pulse_duration, an integer value of cells_per_beam can be specified, and the range resolution and pulse_duration computed from this.

↓ decreases

dec_rate decision_rate

det_prob detection_probability

FAR false-alarm_rate

resol resolution

Effect of probability of false alarm

PFA ↑ =L=> FAR ↑ ==> FAR_units ↑

PFA ↑ ==> det_prob ↑ =L=> hit_rate ↑ ==> hit_rate_units ↓.

Effect of scan rate

scan_rate ↑ =L=> dec_rate ↑ =L=> FAR ↑ ==> FAR_units ↑

scan_rate ↑ =L=> det_prob ↓ =L=> hit_rate ↓ ==> hit_rate_units ↑

scan_rate ↑ =L=> hit_rate ↑ ==> hit_rate_units ↓.

Effect of pulse duration

pulse_duration ↑ =L=> range_resol ↑ ==> target_resol_units ↑

pulse_duration ↑ =L=> range_resol ↑ ==> blind_range_units ↑

pulse_duration ↑ =L=> range_resol ↑ =L=> cells_per_beam ↓ =L=>
dec_rate ↓ =L=> FAR ↓ ==> FAR_units ↓

For pulse_duration > 2E-6:

pulse_duration ↓ ==> PRF ↑ ==> detection_probability ↓
==> hit_rate_units ↑

Alternative to above:

cells_per_beam ↑ =L=> range_resol ↓ ==> target_resol_units ↓

cells_per_beam ↑ =L=> range_resol ↓ ==> blind_range_units ↓

cells_per_beam ↑ =L=> dec_rate ↑ =L=> FAR ↑ ==> FAR_units ↑

For cells_per_beam > 154:

cells_per_beam ↑ ==> PRF ↑ ==> detection_probability ↓
==> hit_rate_units ↓

PERFORMANCE MEASURE BOUNDARIES

In the example problem chosen, two of the parameters, scan_rate and pulse_duration (equivalently, cells_per_beam), have minimum and maximum values. The range of PFA is less obvious except, of course, that it ranges from zero to unity. Within the minimum and maximum parameter values, there may be values that result in a performance measure exceeding its maximum. An example of this is shown in figure 11, where scan_rate is fixed at 12 rpm and performance measures are computed as PFA and cells_per_beam are varied. Low values of PFA result in a small detection_probability and therefore in excessive hit_rate_units, and high values result in a too-high false-alarm_rate. Low values (less than 35) of cells_per_beam result in excessive target_resolution_units in the case where the target_count is 30.

In our simple example, these boundaries can be computed. If we had chosen a realistic probability distribution for detection probability, or had considered an agile-beam problem, this would have been impractical or impossible. Therefore, we will assume that our learning system does not have access to an inverse formula for finding PFA as a function of detection_probability, cells_per_beam, and scan_rate. Having the learning system discover the boundary when hit_rate_units exceed 100 permits experiments with techniques applicable to other boundary-finding situations. On the other hand, finding PFA as a function of cells_per_beam and scan_rate, for FAR_units = 100, can be reasonably done with a formula in many radar optimization cases, so a formula will be used for this boundary. This maximum value of PFA is

max_PFA = 0.0070872 / (scan_rate * cells_per_beam) .

In the simple method described in appendix A, the user specifies the coarse-scan parameters. The ones used there are the following.

scan_rate = 6, 8, 10, 12, 14, 16, 18

cells_per_beam = 35, 100, 200, 300, 400, 500, 600

PFA = coarse_factor * max_PFA,

where coarse_factor = $1, \sqrt{0.1}, 0.1, \sqrt{0.01}, 0.01, \sqrt{0.001}, 0.001$.

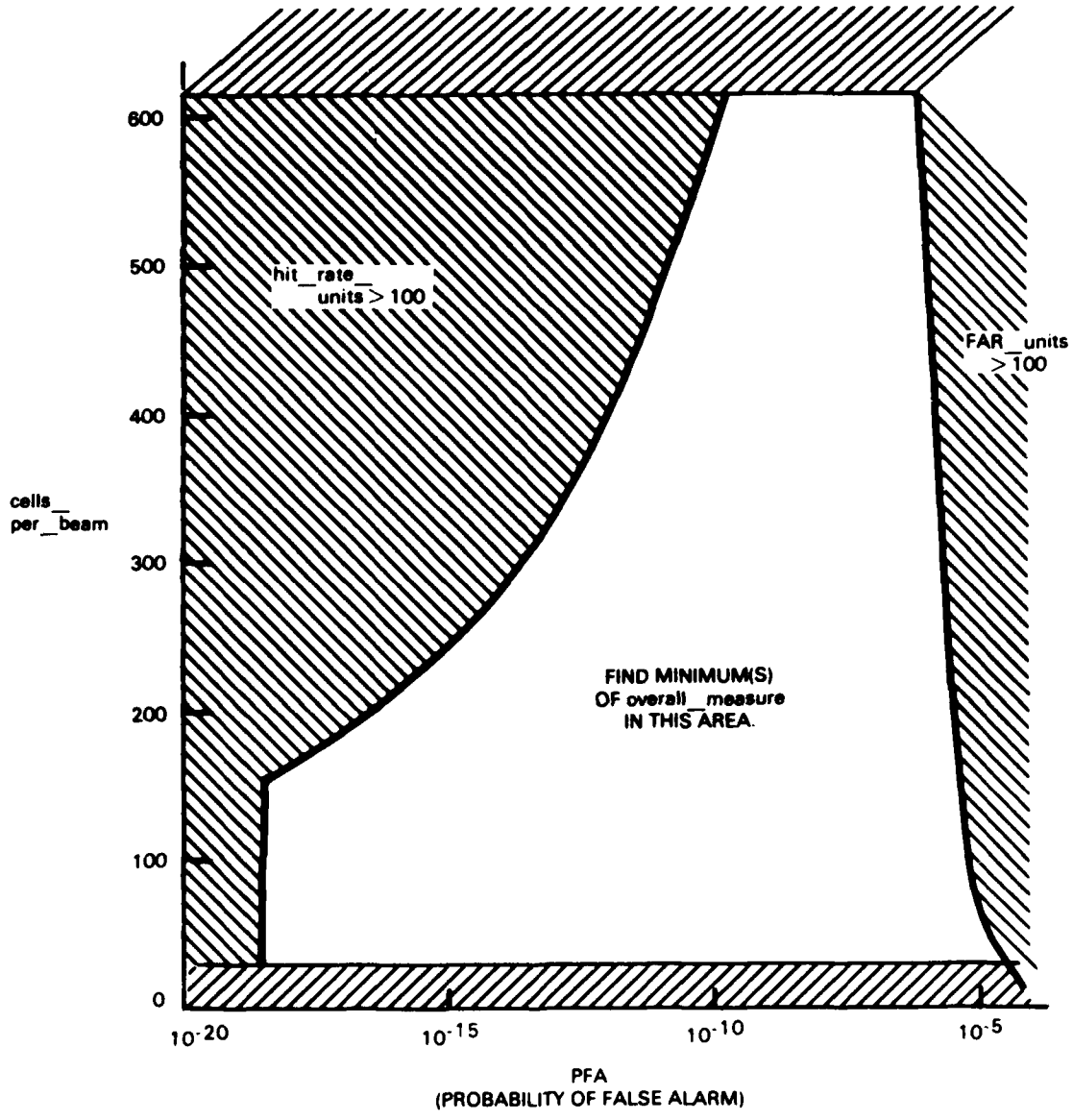


Figure 11. Optimization problem for a 12-rpm scan rate. The value of cells_per_beam varies from 31 to 618 because of constraints on pulse_duration. For cells_per_beam > 154, the PRF increases with cells_per_beam, causing the output signal_to_noise ratio to decrease, and thus the hit rate to decrease for fixed PFA. For target_count = 30, the target_resolution_units exceed 100 if cells_per_beam < 35.

If a system is to be used for several optimization problems, it would be practical to build in the capability of automatically computing maximum and minimum values as well as an appropriate increment. Furthermore, the increment for each parameter could vary, depending on previous results. The user may have to specify whether the parameter should be incremented linearly, logarithmically (as with PFA), or by some other method.

SAVINGS IN COMPUTATION TIME

Optimization of the parameters for the problem described would require much less effort and computation time if, instead of using a learning system, we used a "fine scan" over promising regions after the coarse scan, or even simply used a fine scan over the entire region. We chose this simple example because experimenting with learning techniques then does not require a computer program for simulating radars and trackers. If a simulator is available, more realistic performance measures can be used, some relating to false-track rate, track quality, and target separation. However, this realism is not needed in the early development of the learning techniques, when the learning techniques are developed to the point that they can greatly cut down on the number of simulations that must be performed. Then they can be combined with a simulator, and used for real applications.

CONCLUSIONS

When we began this project as a small effort in FY85, we expected to code and experiment with some learning techniques applicable to systems as complex as agile-beam radars. During that year, we became aware of the enormous complexity and detail needed even for a simple problem, and determined that we did not have the time or adequate computing capability for realistic experiments. We decided that a considerable amount of knowledge could be gained just by designing a system in generic object-based code. We completed such a design in FY86 (although sketchy in parts), and we hope later, under other funds, to implement the example described and to expand the implementation to more complex applications.

The learning techniques proposed for the example problem of a simple mechanical-scan radar are based on humanlike reasoning processes. The problem was modeled as one of optimizing parameters, with the conversion of results into parameter-control rules occurring after the parameter optimization process. It was pointed out that in the case of an agile-beam radar system, one could formulate rules containing parameters and then apply parameter optimization procedures to the rules' parameters. In practice, the rules' structures would also need to be optimized, and the problem is larger than that of parameter optimization.

In FY86, this project was expanded to include investigating other learning techniques that might be applicable to system optimization problems or to pieces of the problem. These techniques differ from those discussed here in that they do not primarily imitate human behavior and in that they directly address the problem of learning rules about optimum parameters. The results of these investigations will be reported on separately. In other work (reference 6) of possible interest to the reader, researchers at the Naval Surface Warfare Center are investigating the use of genetic algorithms to refine a combat system's doctrine rules.

Our investigations have substantiated our original supposition that a learning system would not be practical unless it was very general and could be applied to a variety of specific problems. Recall that figure 1 shows an "initial data/knowledge acquisition system" that understands the basics of the generic radars and learns from the user the specifications of the particular system and the environment. The techniques used in the simple example in appendix A are essentially applicable to very general parameter optimization problems having a similar definition of "optimum."

REFERENCES

1. McArthur, D., and P. Klahr, The ROSS Language Manual, Rand Note N-1854, Sep 1982.

2. Langley, P.W., Rediscovering Physics with BACON.3, IJCAI-6, vol. 1, 505-507, 1977.

3. Cohen, P.R., and E.A. Feigenbaum, The Handbook of Artificial Intelligence, vol III, HeurisTech Press, Stanford, CA, 1982.

4. Robson, D., Object-Oriented Software Systems, Byte, vol. 6, no. 8, pp 74-86, Aug 1981.

5. Robson, D., and A. Goldberg, The Smalltalk-80 System, Byte, vol. 6, no. 8, pp 36-48, Aug 1981.

6. Kuchinski, M.J., Battle Management Systems Control Rule Optimization Using Artificial Intelligence, NSWC/MP-84-329, Jun 1985.

APPENDIX A:
AN EXPERIMENTAL SYSTEM IN GENERIC OBJECT-BASED CODE

Notes:

1. Prefix $ marks an object whose value can vary with each test or each environment. (Value is one "attribute" of certain objects.)

2. Prefix & marks the value of an object. If <name> is a fixed parameter, let &<name> be the value of <name>. If <name> is a variable parameter or an intermediate measure, let &<name> be the value of $<name>.

3. Formulas are listed just below where first referred to, rather than in the section on Initial Knowledge.

4. <> represents a value to be assigned by the action of some object. Angle brackets also enclose a description of what is in that slot.

5. Two hyphens (--) precede lines of comment that appear in the code but are not part of it.

TOP-LEVEL OBJECT

```
object
    offspring = (initial_knowledge fake_simulator performance_monitor learner)
```

INITIAL KNOWLEDGE

```
-- See figure A1.

initial_knowledge
    parent = object
    offspring = (connection specification)
```

```
connection
      parent = initial_knowledge
      offspring = (formula dependency)

formula
      parent = connection
      offspring = (formula_1 formula_2 ...)

-- Formulas appear where first referred to.
dependency
      parent = connection
      offspring = (exact_dependency approx_dependency unknown_dependency)

-- Dependencies are treated later.
specification
      parent = initial_knowledge
      offspring = (optimization_problem environment_spec radar_spec
performance_spec text)

text
      parent = specification
      offspring = (goal_step_1 goal_step_2 PRF_formula spread_formula)

-- Text is sometimes used here to describe initial knowledge of algorithms
-- or procedures to be called by the behavior of objects in other subsystems.
-- A subroutine, LISP function, or other form would be used in a specific
-- object-based system.  The text shown (in quotes in later examples) could
-- be saved under the attribute "description," and the parent would become
-- "subroutine" or other.

                              -- optimization_problem --
-- See figure A2.

optimization_problem
      parent = initial_knowledge
```

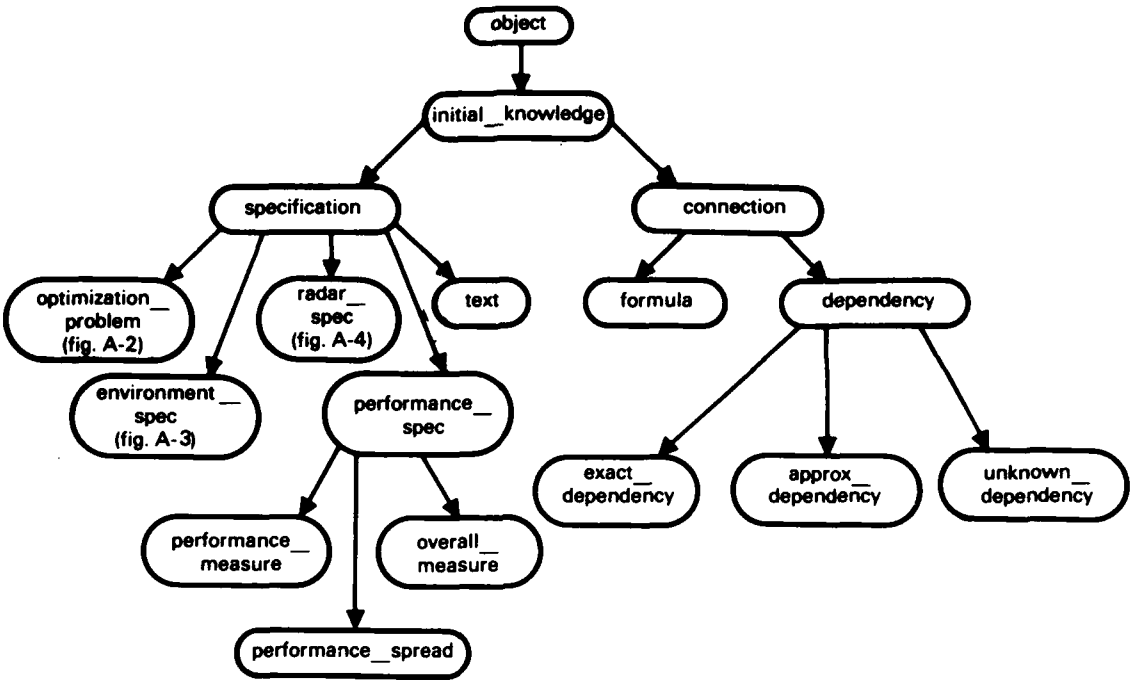


Figure A-1. Top-level objects under initial_knowledge.

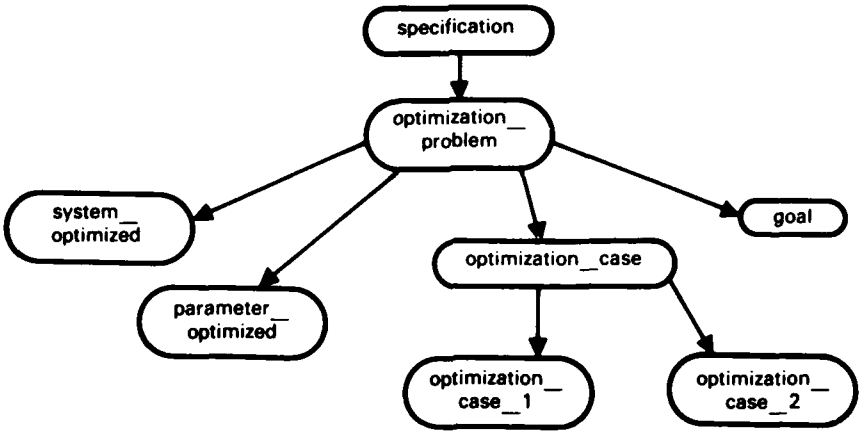


Figure A-2. Specification of the optimization_problem, under initial knowledge.

```
    offspring = (system_optimized parameter_optimized optimization_case
goal)

system_optimized
    parent = optimization_problem
    type = radar
    system_spec = radar_spec

parameter_optimized
    parent = optimization_problem
    parameters = (scan_rate pulse_duration PFA)
    varies_with = optimization_case

-- See the explanation below (under "radar_spec") of why the variable
-- parameters peak_power and PRF are not involved in the optimization process.

optimization_case
    parent = optimization_problem
    offspring = (optimization_case_1 optimization_case_2)

optimization_case_1
    parent = optimization_case
    environment = environment_1
    radar_mode = mode_1

optimization_case_2
    parent = optimization_case
    environment = environment_2
    radar_mode = mode_2
goal
    parent = optimization_problem
    varies_with = optimization_case
    value = (goal_step1 goal_step2)
```

```
goal_step1
      parent = text
      value = "Find the parameter set such that overall_measure M is
               minimized, and denote it s1."

goal_step2
      parent = text
      value =   "To balance component performance measures, find parameter set s2
               such that performance_spread S is reduced at a small sacrifice in
               overall_measure M.  In particular, find s2 such that S(s2) is the
               minimum S within the constraints:
                    M(s2) / M(s1) < 1.1
               and S(s1) - S(s2) > 4 * (M(s2) - M(s1)) / M(s1). "

-- Goal_step1 and goal_step2 are used in the design of the "optimizer"
-- part of the learner.  The algorithm is just a simple example.

-- The performance measures, overall_measure, and performance_spread are
-- defined under "performance_spec."

                                   -- environment_spec --
-- See figure A3.

environment_spec
      parent = specification
      offspring = (target_type environment)

target_type
      parent = environment_spec
      offspring = standard_target  -- Normally, there would be several types.
standard_target
      parent = target_type
      category = surface
      cross_section = 10
```

```
        cross_section_units = meters_squared
        range = 10
        range_units = nautical_miles

environment
        parent = environment_spec
        offspring = (environment_1 environment_2)

environment_1
        target_count = 1
        sea_state = 1
        weather = clear

environment_2
        target_count = 30
        sea_state = 1
        weather = clear

                                -- radar_spec --

-- See figure A4.

radar_spec
        parent = specification
        offspring = (radar_overview parameter mode intermediate_measure)

radar_overview
        parent = radar_spec
        name = radar_simplistica
        scan_type = mechanical_scan
        beam_type = fan
        function = surface_search
        processing = noncoherent_pulse_integration
```

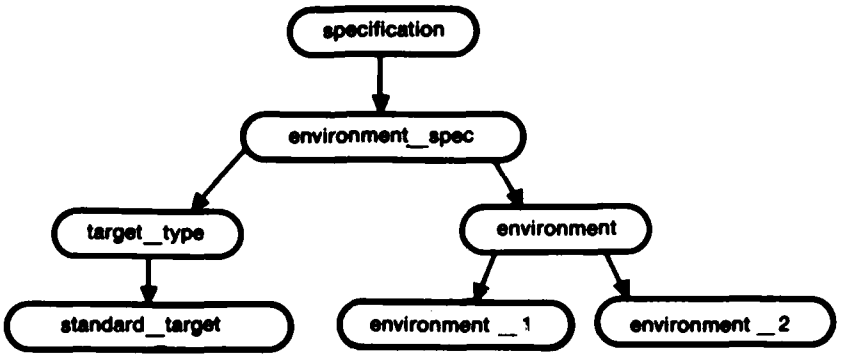


Figure A-3. Specification of the environment, under initial_knowledge.

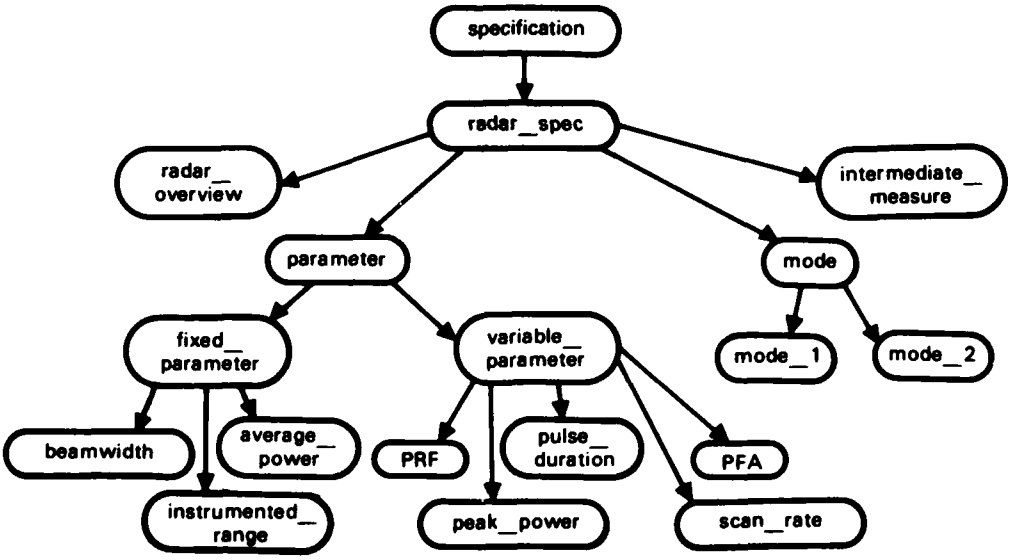


Figure A-4. Specification of the radar, under initial_knowledge.

```
                                -- parameter --

parameter
        parent = radar_spec
        offspring = (fixed_parameter variable_parameter)

fixed_parameter
        parent = parameter
        offspring = (beamwidth instrumented_range average_power)

beamwidth
        parent = fixed_parameter
        value = 1.5
        units = degrees_azimuth

instrumented_range
        parent = fixed_parameter
        value = 25
        units = nautical_miles

average_power
        parent = fixed_parameter
        value = 10
        units = watt-seconds
        -- average_power = peak_power * pulse_duration * PRF

variable_parameter
        parent = parameter
        offspring = (PRF peak_power pulse_duration scan_rate PFA)

-- We are assuming that energy consumption is exactly proportional to
-- peak_power, PRF, and pulse_duration; and detection performance is a
-- function of energy_per_pulse and pulses_per_beam.  Since average_energy is
-- a fixed parameter, the variable parameters peak_power and PRF are not
-- varied independently in the optimization_process; they are determined by
```

```
-- the following rule:  Use the smallest PRF such that (1) PRF >= 500;
-- (2) pulses_per_beam = integer; -- and (3) peak_power =< 1E4 watts.  An
-- algorithm for this follows.

PRF_formula
      parent = text
      value = "Let NLI = next lower integer.
                If pulse_duration > 2E-6,
                  let i = 1 + NLI(125 / scan_rate);
                otherwise,
                  let i = 1 + NLI(1 / (4E3 * pulse_duration * scan_rate))
                Let PRF = 4 * i * scan_rate."

PRF
      parent = variable_parameter
      description = pulse_repetition_frequency
      varies_with = mode
      constraint = (integer & pulses_per_beam)
      formula = formula_1
      value_type = integer
      min_value = 500
      max_value = 2100
      units = pulses_per_second

formula_1
      parent = formula
      quantity = PRF
      calls = (pulse_duration scan_rate)
      output = PRF_formula

-- For a 1.5-degree beamwidth, PRF's constraint is satisfied if PRF is a
-- multiple of 4 * scan_rate.

peak_power
      parent = variable_parameter
```

```
        varies_with = mode
        formula = formula_2
        value_type = continuous
        max_value = 1.0E4
        units = watts

formula_2
        parent = formula
        quantity = peak_power
        calls = (&average_power &PRF & pulse_duration)
        output = &average_power / (&PRF * &pulse_duration)

pulse_duration
        parent = variable_parameter
        constraint = (integer &cells_per_beam)
        varies_with = mode
        value_type = continuous
        min_value = 0.5E-6
        max_value = 1E-5
        formula = formula_3
        units = seconds
        -- Since the pulse_duration has a constraint that the cells_per_beam
        -- must be an integer, it is easiest to fix the intermediate measure
        -- cells_per_beam, compute the intermediate measure range_resolution
        -- (using the alternative version of formula_5), and then compute
        -- pulse_duration (using formula_3).

formula_3
        parent = formula
        quantity = pulse_duration
        calls = &range_resolution
        output = 1.236E-5 * &range_resolution

scan_rate
        parent = variable_parameter
```

```
        varies_with = mode
        value_type = continuous  -- integer preferred
        min_value = 6
        max_value = 18
        coarse_value = (6 8 10 12 14 16 18)
        -- A coarse_valley will be built using only the coarse values of the
        -- parameter.
        units = revolutions_per_minute

PFA
        parent = variable_parameter
        description = prob_of_false_alarm_per_resolution_cell_decision
        varies_with = mode
        category = indirect
        value_type = continuous
        min_value = 0
        max_value = 1
        coarse_value = computed
        -- Coarse_scanner computes &coarse_scan_PFA.
        units = probability

                                -- mode --

mode
        parent = radar_spec
        offspring = (mode_1 mode_2)
        -- Could be more than two if multiple usable valleys exist.
        varies_with = environment

mode_1
        parent = mode
        employment = environment_1
        PRF = <>
        peak_power = <>
        pulse_duration = <>
```

```
        scan_rate = <>
        PFA = <>
        -- Final values correspond to an optimum_set.

mode_2
        parent = mode
        employment = environment_2
        -- Same attributes as mode_1

                                -- intermediate_measure --

intermediate_measure
        parent = radar-spec
        offspring = (pulses_per_beam range_resolution cells_per_beam
                     decision_rate false-alarm_rate energy_per_pulse
                     detection_probability hit_rate)

pulses_per_beam
        parent = intermediate_measure
        type = predetermined
        formula = formula_4
        units = pulses_per_beam_per_scan
        -- The value of pulses_per_beam affects detection probability,
        -- but is not used directly.

formula_4
        parent = formula
        quantity = pulses_per_beam
        calls = (&PRF &scan_rate)
        output = &PRF / (4 * &scan_rate) -- for a 1.5-degree beamwidth

range_resolution
        parent = intermediate_measure
        type = predetermined
```

```
        formula = formula_5
        units = nautical_miles

formula_5
        parent = formula
        quantity = range_resolution
        calls = &pulse_duration
        output = 0.8094E5 * &pulse_duration

-- Next formula allows integer value of cells_per_beam to be specified.

formula_5  -- alternative
        parent = formula
        quantity = range_resolution
        calls = &cells_per_beam
        output = 25 / &cells_per_beam  -- for a 25-nmi instrumented range

cells_per_beam
        parent = intermediate_measure
        description = resolution_cells_per_antenna_beam
        type = predetermined
        value_type = integer
        min_value = 31
        max_value = 618
        coarse_value = (35 100 200 300 400 500 600)
        formula = formula_6
        units = cells_per_beam

-- Next formula not necessary if previous formula employed.

formula_6
        parent = formula
        quantity = cells_per_beam
        calls = &range_resolution
        output = 25 / &range_resolution  -- for a 25-nmi instrumented range
```

```
decision_rate
        parent = intermediate_measure
        description = signal_or_noise_decisions_per_second
        type = predetermined
        formula = formula_7
        units = cells_per_second

formula_7
        parent = formula
        quantity = decision_rate
        calls = (&scan_rate &cells_per_beam)
        output = 4 * &scan_rate * &cells_per_beam
        -- for a 1.5-degree beamwidth

false-alarm_rate
        parent = intermediate_measure
        type = probabilistic
        formula = formula_8
        units = false_alarms_per_hour

formula_8
        parent = formula
        quantity = false-alarm_rate
        calls = (&decision_rate &PFA)
        output = 3600 * &decision_rate * &PFA

energy_per_pulse
        parent = intermediate_measure
        type = predetermined
        formula = formula_9
        units = watt-seconds
        -- The value of energy_per_pulse affects detection_probability, but is
        -- not used directly.
```

```
formula_9
        parent = formula
        quantity = energy_per_pulse
        calls = &PRF
        output = 10 / &PRF
        -- = &peak_power * &pulse_duration

detection_probability
        parent = intermediate_measure
        type = probabilistic
        formula = formula_10
        units = probability_per_scan

formula_10
        parent = formula
        quantity = detection_probability
        calls = (&PFA &scan_rate &PRF)
        output = exp((ln &PFA) / (2500 / SQRT(&scan_rate * &PRF) + 1))
            -- Approximated with Rayleigh distribution.
            -- For standard_target only  (10 nmi, 10 m²).
            -- Assumes noncoherent integration:  Output SNR =
            -- constant * energy_per_pulse * SQRT(pulses_per_beam)

hit_rate
        parent = intermediate_measure
        description = average_number_detections_of_target_per_minute
        type = probabilistic
        formula = formula_11
        units = hits_per_minute

formula_11
        parent = formula
        quantity = hit_rate
        calls = (&scan_rate &detection_probability)
        output = &scan_rate * &detection_probability
```

```
                        -- performance_spec --

-- See figure A1.

performance_spec
        parent = specification
        offspring = (performance_measure overall_measure performance_spread)

performance_measure
        parent = initial_knowledge
        varies_with = scenario
        offspring = (FAR_units hit_rate_units target_resolution_units
                     blind_range_units)

FAR_units
        parent = performance_measure
        formula = formula_12
        units = rejection_units
        max_units_allowed = 100

formula_12
        parent = formula
        quantity = FAR_units
        calls = &false-alarm_rate
        output = 120 * exp(2 * ((log &false-alarm_rate) - 2.1))

hit_rate_units
        parent = performance_measure
        formula = formula_13
        units = rejection_units
        max_units_allowed = 100

formula_13
        parent = formula
        quantity = hit_rate_units
```

```
        calls = &hit_rate
        output = 115 * exp(-0.7 * (&hit_rate - 3))

target_resolution_units
        parent = performance_measure
        formula = formula_14
        units = rejection_units
        max_units_allowed = 100

formula_14
        parent = formula
        quantity = target_resolution_units
        calls = (&range_resolution &target_count)
        output = (50 + 3 * &target_count) * &range_resolution

blind_range_units
        parent = performance_measure
        formula = formula_15
        units = rejection_units
        max_units_allowed = 100

formula_15
        parent = formula
        quantity = blind_range_units
        calls = &range_resolution
        output = 30 * &range_resolution

overall_measure
        parent = performance_spec
        varies_with = simulation_test
        formula = formula_16
        units = rejection_units
```

```
formula_16
      parent = formula
      quantity = overall_measure
      calls = <offspring of $performance_measure>
      output = <sum of offspring of $performance_measure>

performance_spread
      parent = performance_spec
      varies_with = simulation_test
      formula = formula_17
      units = normalized_rejection_units

formula_17
      parent = formula
      quantity = performance_spread
      calls = (<offspring (Mi) of performance_measure>
                <offspring of $performance_measure>)
      output = spread_formula

spread_formula
      parent = text
      value = "performance_spread = max{qi} - min{qi}, where
               qi = &Mi / (max_units_allowed of Mi)
               q1 = &FAR_units / 100
               q2 = &hit_units / 100
               q3 = &target_resolution_units / 100
               q4 = &blind_range_units / 100 "

                              -- dependencies --

approx_dependency
      parent = dependency
      offspring = (approx_dependency_1 ...)
```

```
unknown_dependency
      parent = dependency
      offspring = (unknown_dependency_1 ...)

approx_dependency_1
      parent = approx_dependency
      type = support
      quantity1 = PFA
      quantity2 = FAR_units

approx_dependency_2
      parent = approx_dependency
      type = against
      quantity1 = PFA
      quantity2 = hit_rate_units

approx_dependency_3
      parent = approx_dependency
      type = support
      quantity1 = scan_rate
      quantity2 = FAR_units

unknown_dependency_1
      parent = unknown_dependency
      type = mixed
      quantity1 = scan_rate
      quantity2 = hit_rate_units
      -- Hit opportunities increase with scan_rate, but detection_probability
      -- decreases.

approx_dependency_4
      parent = approx_dependency
      type = against
      quantity1 = cells_per_beam
      quantity2 = target_resolution_units
```

```
approx_dependency_5
      parent = approx_dependency
      type = against
      quantity1 = cells_per_beam
      quantity2 = blind_range_units

approx_dependency_6
      parent = approx_dependency
      type = support
      quantity1 = cells_per_beam
      quantity2 = FAR_units

approx_dependency_7
      parent = approx_dependency
      type = against
      constraint = (less_than 154 cells_per_beam)  -- from PRF_formula
      quantity1 = cells_per_beam
      quantity2 = hit_rate_units

-- Dependencies involving intermediate measures could also be given and,
-- in fact, could have been used to automatically generate all those
-- given above.
```

FAKE SIMULATOR

```
-- See figure A5.

fake_simulator
      parent = object
      offspring = (radar_simulator $target_count)

$target_count
      parent = fake_simulator
      units = targets
      value = <1 or 30>
```

```
radar_simulator
        parent = fake_simulator
        offspring = ($test_parameter $intermediate_measure)

                                -- $test_parameter --

$test_parameter
        parent = radar_simulator
        offspring = ($PRF $peak_power $pulse_duration $scan_rate $PFA)
        test = <integer>
        -- Attribute "test" has same value for all objects having this
        -- attribute.  The value is incremented by 1 after the outcome
        -- of a test is used to produce a new parameter set.

$PRF
        parent = $test_parameter
        value = <>

$scan_rate
        parent = $test_parameter
        value = <>

$pulse_duration
        parent = $test_parameter
        value = <>

$peak_power
        parent = $test_parameter
        value = <>

$PFA
        parent = $test_parameter
        value = <>
```

-- $intermediate_measure --

```
$intermediate_measure
      parent = radar_simulator
      offspring = ($pulses_per_beam $range_resolution $cells_per_beam
                  $decision_rate $false-alarm_rate $energy_per_pulse
                  $detection_probability $hit_rate)
      test = <integer>

$pulses_per_beam
      parent = $intermediate_measure
      value = <>
      -- Value for information only, since not used in calculations.

$range_resolution
      parent = $intermediate_measure
      value = <>

$cells_per_beam
      parent = $intermediate_measure
      value = <>

$decision_rate
      parent = $intermediate_measure
      value = <>

$false-alarm_rate
      parent = $intermediate_measure
      value = <>

$energy_per_pulse
      parent = $intermediate_measure
      value = <>
```

```
$detection_probability
      parent = $intermediate_measure
      value = <>

$hit_rate
      parent = $intermediate_measure
      value = <>
```

PERFORMANCE MONITOR

```
-- See figure A6.

performance_monitor
      parent = object
      offspring = ($performance_measure $overall_measure)

$performance_measure
      parent = performance_monitor
      offspring = ($FAR_units $hit_rate_units $target_resolution_units
                   $blind_range_units)
      test = <integer>

$FAR_units
      parent = $performance_measure
      value = <>

$hit_rate_units
      parent = $performance_measure
      value = <>

$target_resolution_units
      parent = $performance_measure
      value = <>
```

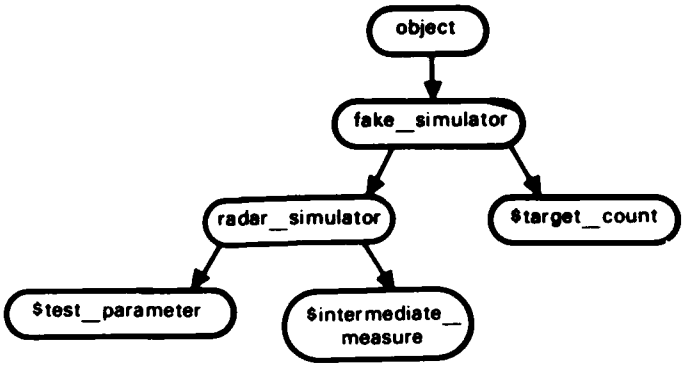


Figure A-5. The fake_simulator. Estimates of detection probability and other measures can substitute for a simulator in early experiments.

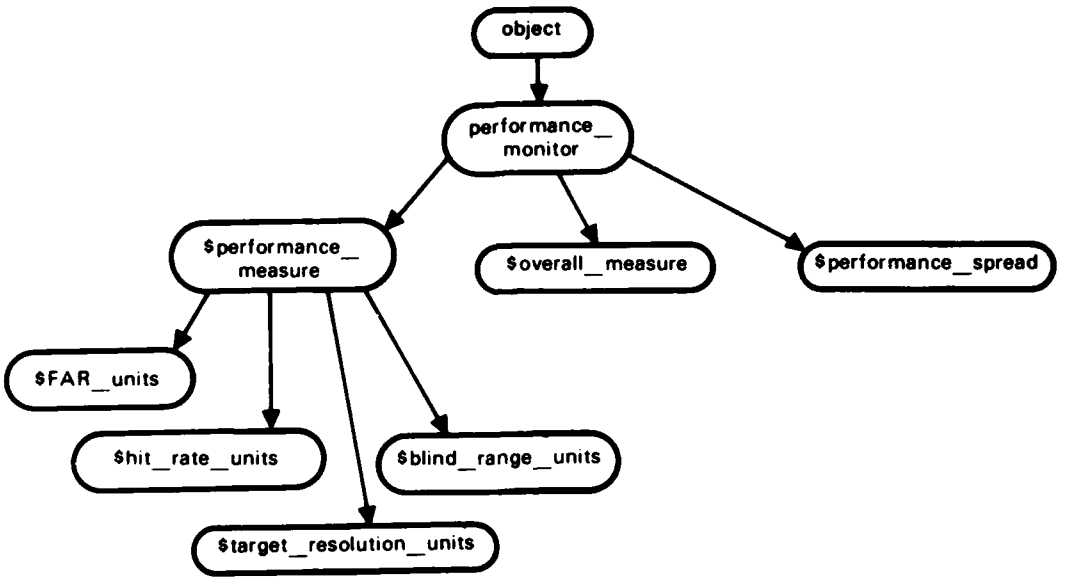


Figure A-6. The performance_monitor. Early experiments can use performance measures that are functions of the intermediate measures computed or estimated by the fake_simulator.

```
$blind_range_units
        parent = $performance_measure
        value = <>

$overall_measure
        parent = performance_monitor
        test = <integer>
        value = <>
        accuracy_measure = <>

$performance_spread
        parent = performance_monitor
        test = <integer>
        value = <>
```

LEARNER

```
-- See figure A7.
learner
        parent = object
        offspring = (next_set history_file valley_finder optimizer)

next_set
        parent = learner
        offspring = (next_set_1 next_set_2 ...)
        -- offspring attributes:
        test = <integer>
        scan_rate = <>
        cells_per_beam = <>
        PFA = <>
        desired_accuracy = <>
        behavior = <Tells fake_simulator when the set is available for
                    the next test.>

-- Receives parameters from valley_finder or optimizer offspring.
```

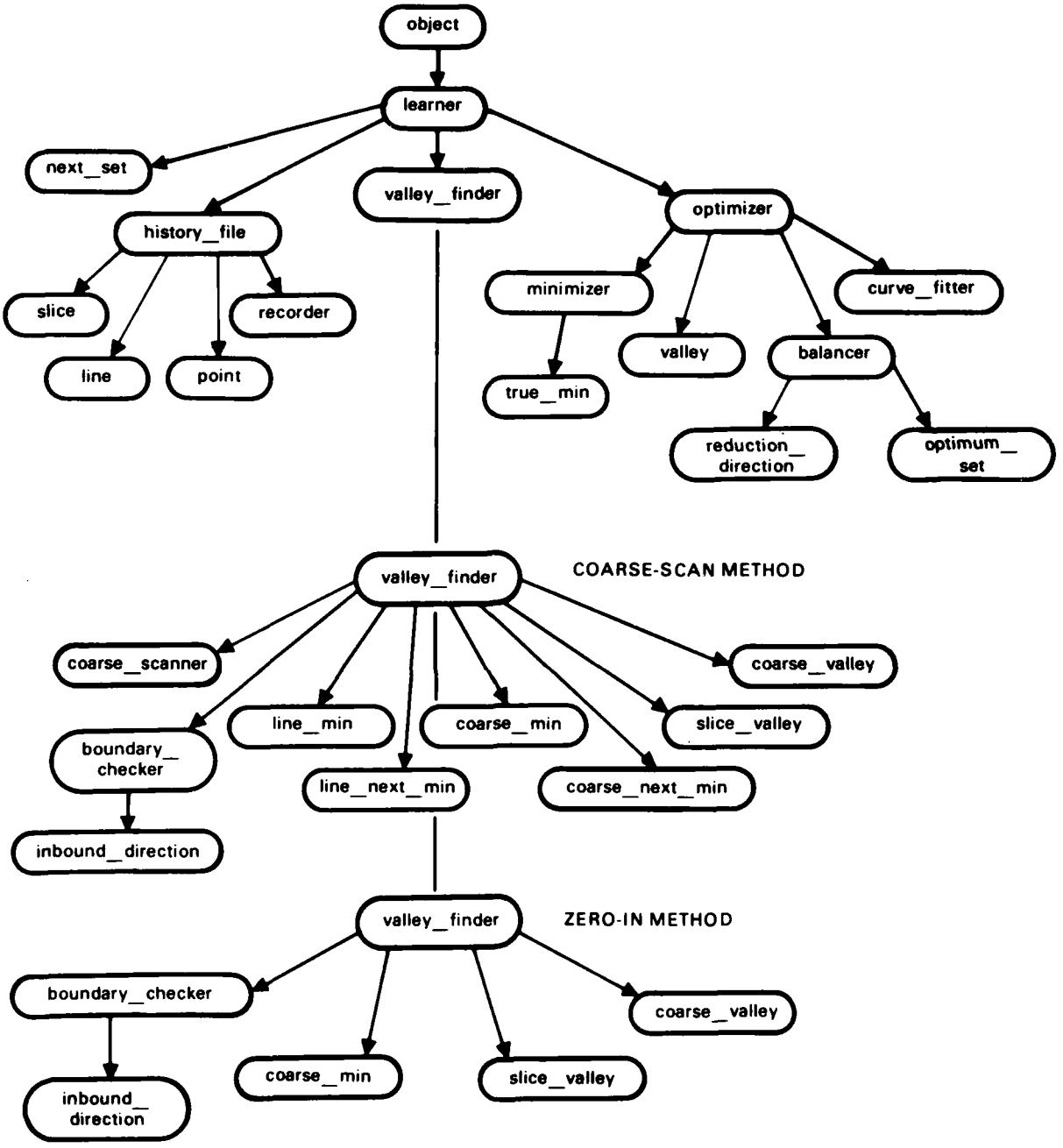


Figure A-7. The learner.

-- history_file --

history_file
    parent = learner
    offspring = (slice line point recorder)

-- See figure A8.

slice
    parent = history_file
    offspring = (slice_1 slice_2 ...)
    scan_rate = <> -- offspring attribute

line
    parent = history_file
    offspring = (line_1 line_2 ...)
    -- offspring attributes:
    slice = <>
    cells_per_beam = <>
    range_resolution = <>
    PRF = <>
    decision_rate = <>
    blind_range_units = <>
    target_resolution_units = <>

-- For this simple case where simulation is not actually performed,
-- each line instantiation can have the attribute target_resolution_
-- units_1 and target_resolution_units_2, corresponding to environment_1 and
-- environment_2. (Otherwise, environment should be an attribute of each
-- slice.) Similarly, there would be two instantiations each of the
-- overall_measure and performance_spread for each point (next object).

point
    parent = history_file
    offspring = (point_1 point_2 ...)

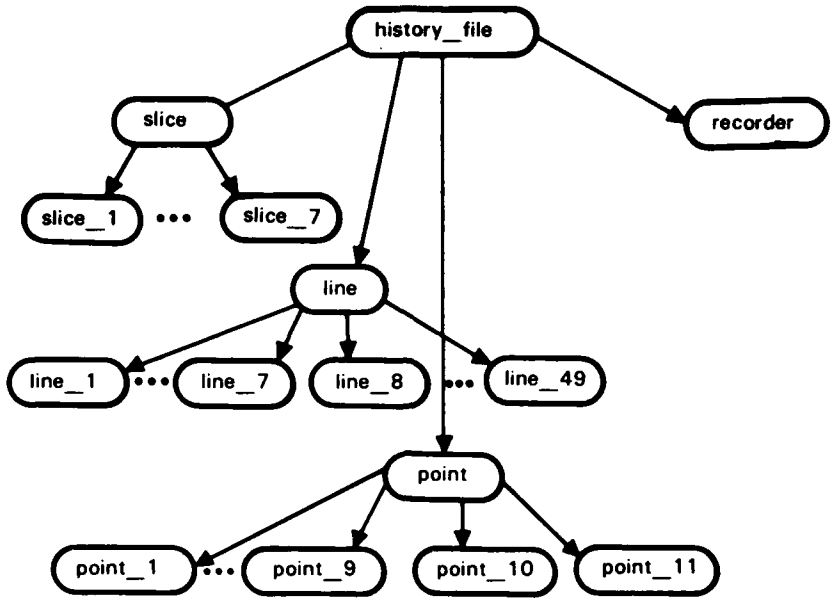


Figure A-8. History_file after coarse scan.

-- offspring attributes:

line = <>

PFA = <>

FAR = <>

detection_probability = <>

hit_rate = <>

FAR_units = <>

hit_rate_units = <>

```
        overall_measure = <>
        accuracy_measure = <>
        performance_spread = <>

-- The mapping of slice, line, and point to scan_rate, cells_per_beam, and
-- PFA is arbitrary; any mapping would work.
-- The following are examples of instantiations for the first point of a
-- coarse scan.  Here, the calculations have been made for both environments;
-- i.e., for target_count = 1 and 30.

slice_1
        parent = slice
        scan_rate = 6

line_1  -- Integer same as value of attribute "test"
        parent = line
        slice = slice_1
        cells_per_beam = 35
        range_resolution = 0.71422857
        PRF = 504
        decision_rate = 840
        blind_range_units = 21.428571
        target_resolution_units_1 = 37.857143
        target_resolution_units_30 = 100

point_1
        parent = point
        line = line_1
        PFA = 3.3749E-5  -- max_PFA
        FAR = 102.056
        detection_probability = 0.79405
        hit_rate = 4.7643
        FAR_units = 100
        hit_rate_units = 33.445663
        overall_measure_1 = 192.731377
```

overall_measure_30 = 254.874234
performance_spread_1 = 0.785714
performance_spread_30 = 0.785714

recorder
parent = history_file
test = <integer>
behavior = <Gets results from fake_simulator and performance_monitor,
and creates slices, lines, and points.>

-- valley_finder --

valley_finder -- coarse-scan version
parent = learner
offspring = (coarse_scanner boundary_checker line_min line_next_min
coarse_min coarse_next_min slice_valley coarse_valley)
behavior = <After a coarse scan, creates line_mins, a coarse_min,
slice_valleys, and a valley. Might also create
line_next_mins, a coarse_next_min, and, around it, an
extra valley. Has further duties (discussed later) if
the valley's type is edge or incomplete.>

coarse_scanner
parent = valley_finder
offspring = ($min_PFA $coarse_value_PFA)
behavior = <Specifies values of next_set for simulator to use, resulting
in successive slices (one per coarse value of scan_rate),
line-by-line (cells_per_beam coarse values). Each line has
a point per coarse_value_PFA.>
<Calls on boundary_checker to avoid going more than one
point beyond the max_allowed_value of a performance
measure. Uses inbound_direction to decide whether to start
a new line or to start a new slice. Deletes any such
inbound_direction when done.>

```
$min_PFA
        parent = coarse_scanner
        description = PFA_giving_100_FAR_units
        varies_with = line
        formula = formula_18
        value = <>

formula_18
        parent = formula
        quantity = $min_PFA
        calls = (&scan_rate &cells_per_beam)
        output = 0.0070872 / (&scan_rate * &cells_per_beam)

$coarse_value_PFA
        parent = coarse_scanner
        formula = formula_19
        value = <>

formula_19
        -- Output is (10**(-i/2)) * max_PFA, for i from 0 to 6.

boundary_checker
        parent = valley_finder
        offspring = inbound_direction
        behavior = <Compares each performance_measure with its max_allowed
                        _value. If exceeded, uses dependencies (between
                        parameters and performance measures) to create an
                        inbound_direction.  Reports back when done.>

inbound_direction
        parent = boundary_checker
        offspring = (inbound_direction_1 inbound_direction_2 ...)
```

```
-- The following is an example of an inbound_direction.

inbound_direction_3
      parent = inbound_direction
      point = point_41
      exceeded = hit_rate_units
      option = ((increase PFA) (mixed scan_rate) (decrease cells_per_beam))

line_min  -- if coarse-scan

      parent = valley_finder
      offspring = (line_min_1 line_min_2 ... )
      -- offspring attributes:
      environment = environment_<1 or 2>
      slice = slice_<integer>
      min_point = point_<integer i>
      left_point = point_<i-1 or i-2>
      right_point = point_<i+1 or i+2>
      outside_point = point_<i-1 or i+1>

-- The "outside_point" is used instead of the left_point or right_point when
-- the min_point is just next to or on an "edge."  The edge may result from a
-- parameter constraint or from exceeding a component measure limit.  The
-- min_point must be inside the permissible area.  If no simulation is
-- performed just outside of an edge with a min_point, the attribute
-- outside_point is created with a value = nil.

-- Next is an example of an instantiation.

line_min_3
      parent = line_min
      line = line_3
      min_point = point_5
      left_point = point_3
      right_point = point_6
```

```
line_next_min  -- if coarse-scan
      -- Same attributes as line_min.
      -- Occurs if another point is deeper than a point in the valley
      -- around the line_min.
      -- Used to see if a second valley exists.

-- The coarse_min is generated from the line_mins over all slices, for each
-- environment.

coarse_min
      parent = valley_finder
      offspring = (coarse_min_1 coarse_min_2)
      -- one for each environment
      -- offspring attributes:
      environment = environment_<1 or 2>
      slice = slice_<integer>
      line = line_<integer>
      -- slice and line attributes are optional
      point = point_<integer>

coarse_next_min  -- if coarse-scan
      -- Has same attributes as coarse_min.
      -- Is deepest point (chosen from line_mins and line_next_mins) outside of
      -- coarse_min's valley and comparable in overall_measure to that valley's
      -- points.  Probably none will exist.

slice_valley
      parent = valley_finder
      offspring = (slice_valley_1 slice_valley_2 ...)
      -- offspring attributes:
      environment = environment_<1 or 2>
      slice = slice_<integer>
      valley_point = (<list of points>)
      min_line = line_<integer i>
      lower_line = line_<i-1 or i-2>
```

upper_line = line_<i+1 or i+2>
outside_line = line_<i-1 or i+1>

-- The "outside_line" is used instead of the upper_ or lower_line when the
-- min_point is on an edge (see the discussion above for outside_point). It
-- has value nil if no simulation or computation is made for that line.
-- The first slice_valley will have coarse_min as its min_point.
-- Figure A9 illustrates a slice_valley, after the coarse scan. The upper or
-- lower line could have contributed a line_next_min rather than a line_min as

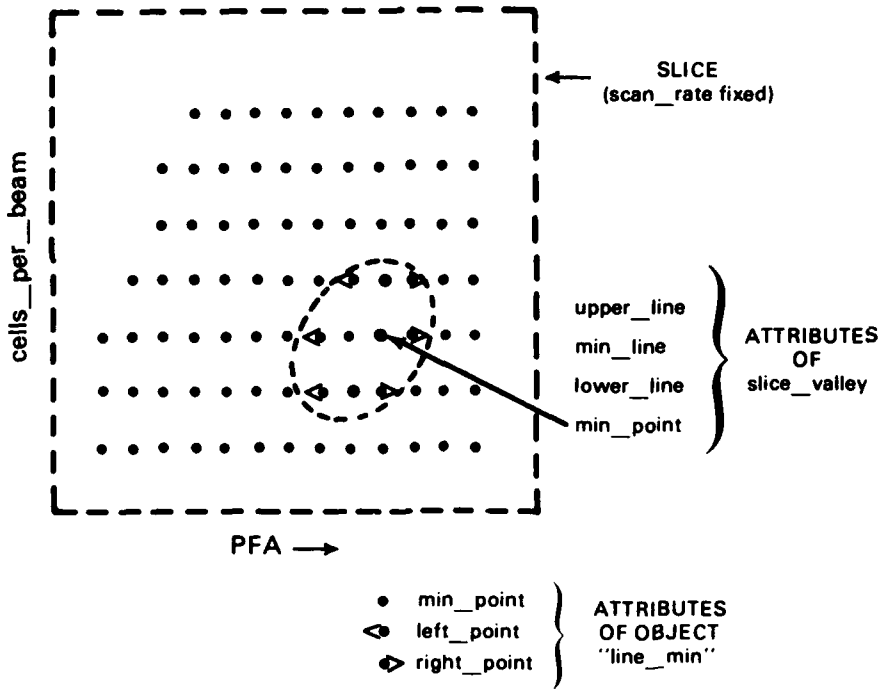


Figure A-9. Example of a valley in one slice, after a coarse scan.

```
-- shown.  Attributes left_point,  right_point, upper_line, and lower_line
-- need be computed only as needed to construct a valley.

coarse_valley
     parent = valley_finder
     offspring = (valley_1 valley_2 ...) -- only 2 for zero-in method
     -- offspring attributes:
     environment = environment_<1 or 2>
     source = <scan_primary, scan_secondary, or zero-in>
     min_point = point_<integer>
     slice_valley = (<3 or more slice valleys>)
     type = <complete, incomplete, or edge>

-- Additional valley_finder behavior:
-- <Classifies valley's type as edge if the min_point is on or next to an edge
-- (an edge as described under "line_min" earlier).  Classifies as incomplete
-- if fails to build a three-slice valley there after the coarse scan.>
-- <If the valley's type = edge, may direct the simulator to perform
-- simulations for additional outside points adjacent to the min_point.  Those
-- outside because a component measure was exceeded will already have had
-- simulations.  Although the outside sets of parameters are not usable, the
-- results are useful for curve fitting.  If the valley's type = incomplete,
-- the valley_finder directs the simulator to improve the accuracy in the area
-- of the min_point.>

-- Figure A10 shows a simple way of creating a valley by using tiers of slice
-- valleys.  Three tiers are shown, but up to five can occur.

valley_finder  -- zero-in version
     parent = learner
     offspring = (boundary_checker coarse_min slice_valley coarse_valley)
     behavior = (<Specifies value of "next_set" for simulator to use.  Uses
                  only coarse values of parameters.  Unless specified
                  otherwise, can begin with a mid-range value of each
                  parameter; e.g., scan_rate = 12, cells_per_beam = 300,
```

PFA = 6E-8.>
<After each simulation, calls on boundary checker and uses its inbound_direction, if needed, to get within bounds. Algorithmically performs 3-dimensional search for coarse_min, using comparisons of overall_measure values.>
<When finds coarse_min, builds slice valleys and a coarse_valley around it and behaves as does coarse-scan version.>)

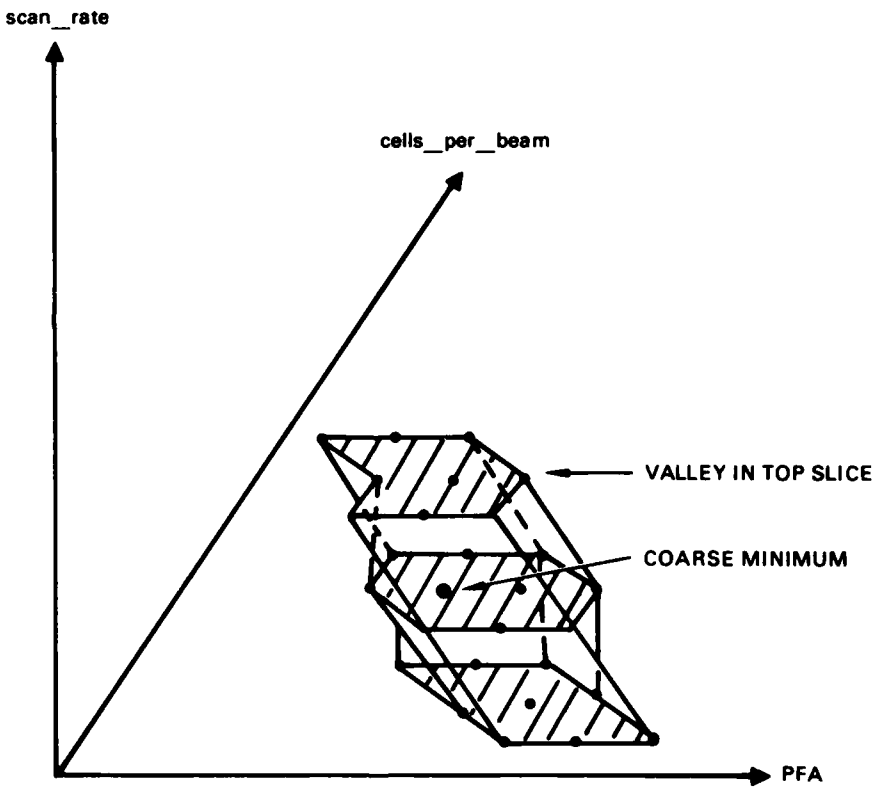


Figure A-10. A simple "three-tier" method of finding a valley.

```
-- When optimization is completed for environment_1, the process for
-- environment_2 would use the results to find an initial parameter set.

slice_valley  -- if zero-in method
      parent = valley_finder
      offspring = (slice_valley_1 slice_valley_2 ... )
      -- offspring attributes:
      environment = environment_<1 or 2>
      slice = slice_<integer>
      valley_point = (<list of points>)
      min_point = point_<integer>

                                    -- optimizer --

optimizer
      parent = learner
      offspring = (minimizer valley balancer curve_fitter)
      cycle = scenario
      behavior = (<Calls on minimizer to find true_min (goal_step_1).>
                  <Calls on balancer to produce optimum_set (goal_step_2).>)

minimizer
      parent = optimizer
      offspring = true_min
      behavior = <Begins with a coarse_valley, and proceeds to accomplish
                  goal_step_1, using the curve_fitter.  Creates a valley from
                  the coarse_valley and new points, while finding the point
                  having the minimum overall_measure.  Successively uses
                  points in the new valley in curve-fitting methods to find
                  that minimum point, and creates the true_min.>

true_min
      parent = minimizer
      offspring = (true_min_1 true_min_2 ...)  -- only 2 if zero-in
      -- offspring attributes:
```

environment = environment_<1 or 2>
valley = valley_<integer>
point = point_<integer>

valley
parent = optimizer
offspring = (valley_1 valley_2 ...) -- only 2 if zero-in
-- offspring attributes:
coarse_valley = coarse_valley_<integer>
new_point = (point_<k> point_<k+1> ...)

balancer
parent = optimizer
offspring = (reduction_direction optimum_set)
behavior = <When the true_min has been created, proceeds to accomplish goal_step_2. Generally adds points to the valley in the process. Initially and after each simulation, uses goal_step_2 algorithm to see if satisfied. If not, creates a "reduction_direction" for the maximum performance measure, using the dependencies involving that measure.>

reduction_direction
parent = balancer
-- same attributes as an inbound_direction
The following is an example of a reduction_direction.

reduction_direction_6
parent = reduction_direction
point = point_46
exceeded = FAR_units
option = ((decrease PFA) (decrease cells_per_beam) (decrease scan_rate))

optimum_set
parent = balancer
offspring = (optimum_set_1 optimum_set_2 ...) -- only 2 for zero-in

-- offspring attributes:
valley = valley_<integer>
point = point_<integer>

curve_fitter
parent = optimizer
offspring = (problem_formulator accuracy_extender adjuster least_squares
n-dim_combiner)
behavior = <When asked to estimate the minimum of a valley, uses its
offspring to do so.>

-- See figure A11.

problem_formulator
parent = curve_fitter
offspring = (edger refiner)
behavior = (<When a coarse_valley has been defined, asks
accuracy_extender to check accuracy and to improve it if
needed.>
<If the accuracy was extended, asks the valley_finder to
redefine the valley, if necessary, since the min_point may
have changed.>
<Next, for each dimension (scan_rate, cells_per_beam, and
PFA), unless min_point is on an edge, asks adjuster to
perform any transformations or normalizations needed before
curve_fitting. Asks the least_squares to estimate the
minimum in that dimension. If the min_point is on an edge
in that dimension, turns control over to edger.>
<When the fitted_min is found in every dimension, asks the
n-dim_combiner to estimate the valley minimum.>)

-- The behavior described is for curve fitting individually in each dimension.
-- The behaviors are different if paraboloids are fitted.

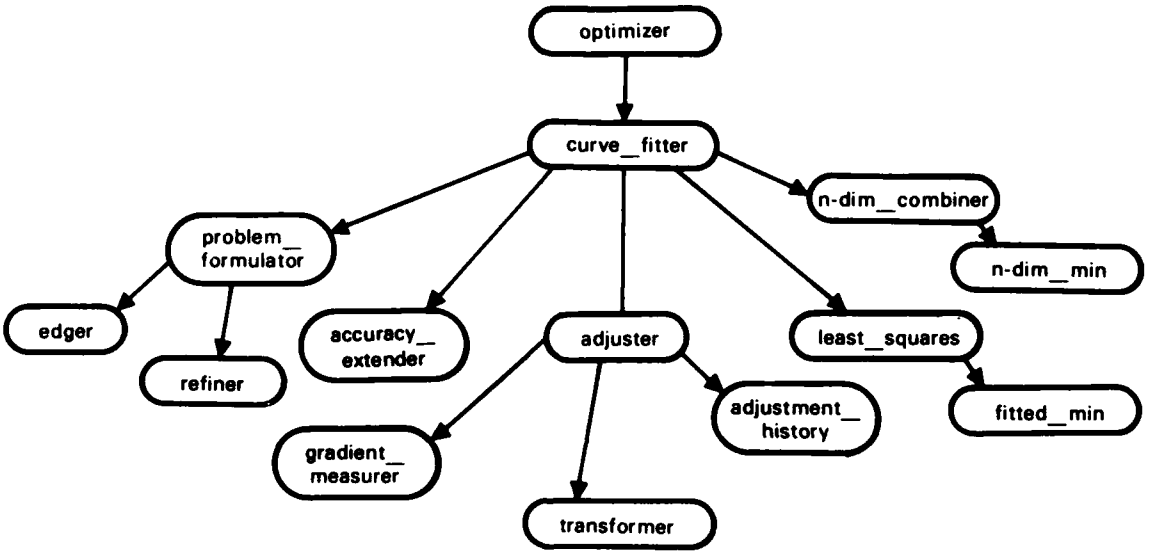


Figure A-11. The curve_fitter.

edger

parent = problem_formulator
behavior = <When the valley's type is edge, for the dimension(s) in which an edge occurs, directs the curve-fitting process to find the fitted_min on the edge.>

refiner

parent = problem_formulator
behavior = <Similar to that above for the problem_formulator, but occurs after simulator provides more points in the area of the previously estimated n-dim_min. Unless simulations are also made on either side of the estimated minimum in each dimension, a paraboloid fitting is needed.>

accuracy_extender

parent = curve_fitter
behavior = <When asked by problem_formulator to improve, if needed, the accuracy in a coarse valley, compares overall_measure differences of points in valleys with the accuracy_measure of the points. Determines if additional accuracy is needed and, if so, asks the simulator, point by point, to extend the accuracy of the overall_measure at that point by some amount. If the original accuracy is very poor, extends accuracy for points just outside the valley. Does this mainly because the valley might shift and because additional points can be used in gradient measurements.>

adjuster

parent = curve_fitter
offspring = (gradient_measurer transformer adjustment_history)
behavior = (<In dimension specified (scan_rate, cells_per_beam, or PFA), calls on gradient_measurer to measure gradients on either side of min_point. Calls on transformer if gradients indicate need. If normalized or transformed, again calls on

gradient_measurer. In general, prepares the problem for the least_squares computations. As a result of final gradient measurements, determines the degree of polynomial needed. (Early experiments would simply use second degree.)>
<When directed by the refiner with additional point in some valley, uses adjustment_history to decide on needed transformation or normalization.>)

gradient_measurer
parent = adjuster
behavior = <Measures slopes from pairs of points around min_point, in specified dimension.>

transformer
parent = adjuster
behavior = <A precurve-fitting transformation or normalization of variables, if needed. Probably not needed except possibly for PFA in this simple application.>

adjustment_history
parent = adjuster
behavior = <Records transformations and normalizations made on a parameter during curve_fitting preparations. Uses this later when curve_fitting with additional points. (Optional, to save repetitious operations.)>

least_squares
parent = curve_fitter
offspring = fitted_min

fitted_min
parent = least_squares
offspring = (fitted_min_1 fitted_min_2 ...)
valley = valley_<>
points_fitted = (point_<> ...)

curve_type = <e.g., parabola>
degree = <>
parameter = <scan_rate, cells_per_beam, or PFA>
parameter_value = <>
measure_estimate = <>

n-dim_combiner
parent = curve_fitter
offspring = n-dim_min
behavior = <Collects fitted_min values in each dimension and produces fitted valley minimum. (See appendix B.)>

n-dim_min
parent = n-dim_combiner
offspring = (n-dim_min_1 ...)
valley = valley_<>
fitted_min = (fitted_min<j> fitted_min_<j+1> fitted_min_<j+2>)
scan_rate = <>
cells_per_beam = <>
PFA = <>
measure_estimate = <> .

APPENDIX B:
CURVE-FITTING METHODS

In practice, an existing computer program for least-squares curve fitting should be selected and adapted to this application. For early experiments, simple parabola-fitting techniques can be implemented. A few basic relationships concerning parabolas and paraboloids are given in this appendix.

Exact Fit to Parabola

The parabola $P(x) = Ax^2 + Bx + C$ fits the three points (x_1, M_1), (x_2, M_2), (x_3, M_3), when

$$A = -[M_1(x_2-x_3) + M_2(x_3-x_1) + M_3(x_1-x_2)] \;/\; (x_1-x_2)(x_2-x_3)(x_3-x_1),$$

$$B = [M_1(x_2^2-x_3^2) + M_2(x_3^2-x_1^2) + M_3(x_1^2-x_2^2)] \;/\; (x_1-x_2)(x_2-x_3)(x_3-x_1),$$

$$C = [x_1^2(M_2x_3-M_3x_2) + x_2^2(M_3x_1-M_1x_3) + x_3^2(M_1x_2-M_2x_1)] \;/$$

$$[x_1^2(x_2-x_3) + x_2^2(x_3-x_1) + x_3^2(x_1-x_2)].$$

The minimum value of $P(x)$ occurs at $x = D$, where $D = -B/2A$. The minimum value is $P(D) = C - B^2/4A$.

If the samples are equally spaced over x, with increment size size s, the coefficients are given by

$$A = (M_1 - 2M_2 + M_3) \;/\; 2s^2,$$

$$B = -[M_1(2x_2+s) - 4M_2x_2 + M_3(2x_2-s)] \;/\; 2s^2,$$

$$C = -[M_1x_2(x_2+s) - 2M_2(x_2^2-s^2) + M_3(x_2-s)x_2] \;/\; 2s^2.$$

Alternative Parabola Form

It is sometimes more convenient to represent the parabola $P(x) = Ax^2 + Bx + C$ in the form

$$P(x) = Ax^2 - 2DAx + D^2A + \text{min},$$

where $B = -2DA$, $C = D^2A + \text{min}$, and $P(D) = \text{min}$. (See figure B1.)

Paraboloid - Two Independent Variables

The paraboloid of the form

$$P(x,y) = Ax^2 + A'y^2 + Bx + B'y + C + C'$$

can also be written as

$$P(x,y) = Ax^2 + A'y^2 - 2DAx - 2D'A'y + D^2A + D'^2A' + \min,$$

where $P(D,D') = \min$. Through this minimum point (D,D') in the $x = D$ plane and $y = D'$ plane, respectively, are the parabolas

$$P(D,y) = A'y^2 - 2D'A'y + D'^2A' + \min$$

and

$$P(x,D') = Ax^2 - 2DAx + D^2A + \min.$$

Paraboloid - k Independent Variables

For k independent variables $V_1, \ldots, V_k$, a paraboloid can be written as

$$P(V_1,\ldots,V_k) = \sum_{i=1}^{k} [A_iV_i^2 - 2D_iA_iV_i + D_i^2A] + \min,$$

where $P(D_1,\ldots,D_k) = \min$.

For $V_j = D_j$, $j \neq i$, we have the parabola

$$P(V_i) = A_iV_i^2 - 2D_iA_iV_i + D_i^2A_i + \min.$$

Paraboloid Minimum

The minimum of a paraboloid can be found or estimated without the complexity of fitting a paraboloid to the sample points. In the case of two independent variables (see fig B2), a parabola can be fit in both dimensions through the minimum measured point (X^*,y^*), and the minimum of the paraboloid (fitting the same points) found from the minimum of the parabolas. If the minimums of the two parabolas are, respectively, $m = P(D,y^*)$ and $m' = P(x^*,D')$, and the measured minimum value is $M^* = P(x^*,y^*)$, then the minimum of the paraboloid $P(x,y)$ is

$$\min = m + m' - M^*.$$

This relationship is exact if the parabolas and the paraboloid exactly fit the sample points. If additional points are used in a least-squares fit (e.g., points diagonal to x^*,y^* when fitting the paraboloid), the relationship is an approximation.

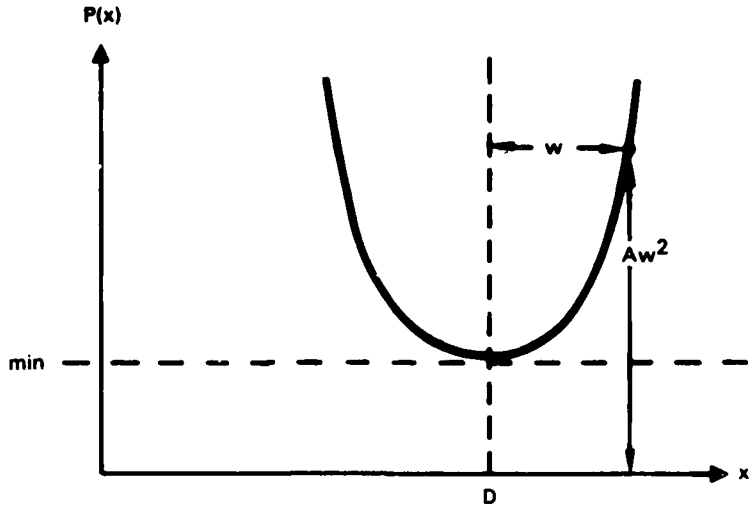


Figure B-1. Parabola $P(x) = Ax^2 - 2DAx + D^2A + min.$

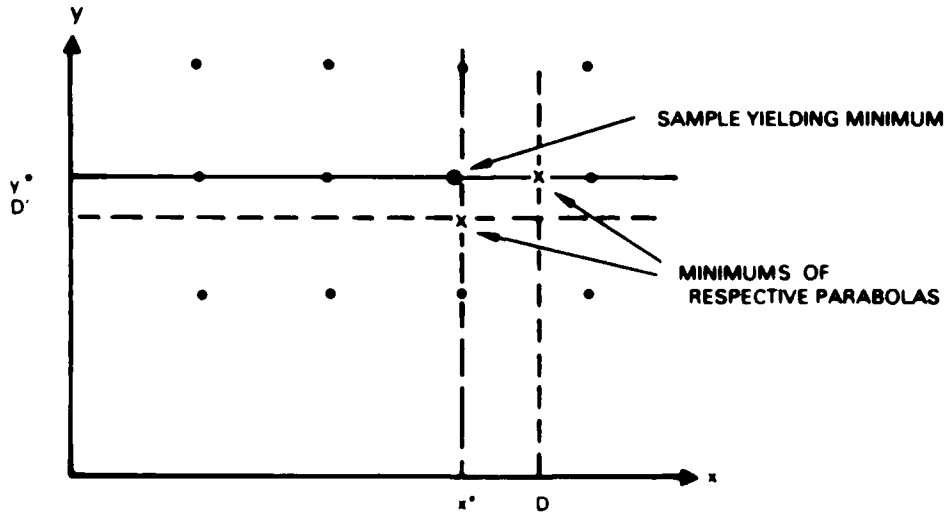


Figure B-2. Parabolas in the $x = x^*$ plane and the $y = y^*$ plane are fitted to the sample points, and their minimums occur at $y = D'$ and $x = D$, respectively. The minimum of the paraboloid fitting these points occurs at (D, D')

For a paraboloid with k independent variables (V_1,....,V_k), the relationship is

$$\text{min} = (m_1 + m_2 + ... + m_k - M^*) / (k-1)$$

where m_i is the minimum of the ith parabola through the minimum measured point (V_1^*, V_2^*,....,V_k^*) and M^* is the measured value at that point.

Precurve-Fitting Operations

A plot of a performance measure versus a parameter x (through a measured minimum value, other parameters held constant) might produce (from simulation) values of performance such as in figure B3. A human could quickly hand fit a curve through the points and estimate the minimum point, while the computer must use more difficult methods. The simplest procedure is to fit a parabola to the minimum and two adjacent points. (Assume this is not a case where the minimum occurs at an edge.) Often this will produce unsatisfactory results, and a higher degree polynomial fitted to additional points is advisable. As a step in selecting the appropriate method, the system could make measurements of the gradients on both sides of the measured minimum and use these in an algorithm to determine the general shape of the curve.

In some cases, the decision should be to transform the variable to produce a curve more easily fitted. Figure B4 shows how a curve of performance measure plotted as a function of log x may be fitted with a parabola, while the same values plotted as a function of x would be difficult to fit.

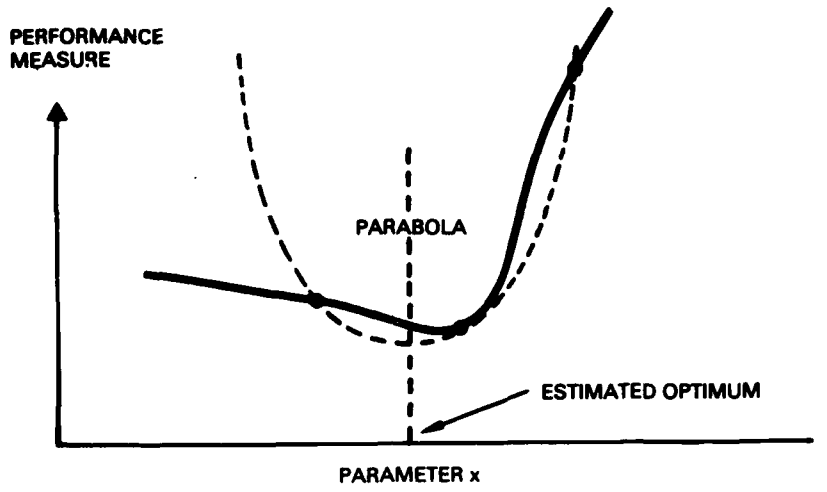


Figure B-3. Example of a hand-fitted curve through a number of points versus a parabola approximation through three points.

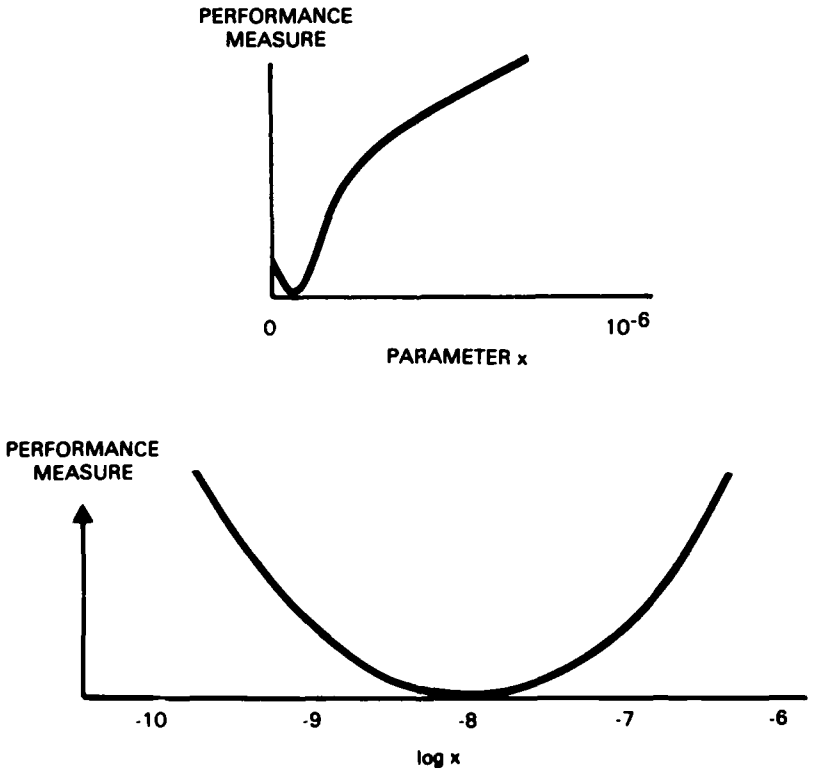


Figure B-4. A simple transformation of variables can sometimes produce data better fitting a parabola. In this example, the transformation is logarithmic.

END

5-87

DTIC